TURING 图灵程序设计丛书

[PACKT] PUBLISHING

[加] Robert Laganière 著　相银初 译

OpenCV计算机视觉编程攻略（第2版）

OpenCV Computer Vision Application Programming Cookbook, Second Edition



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

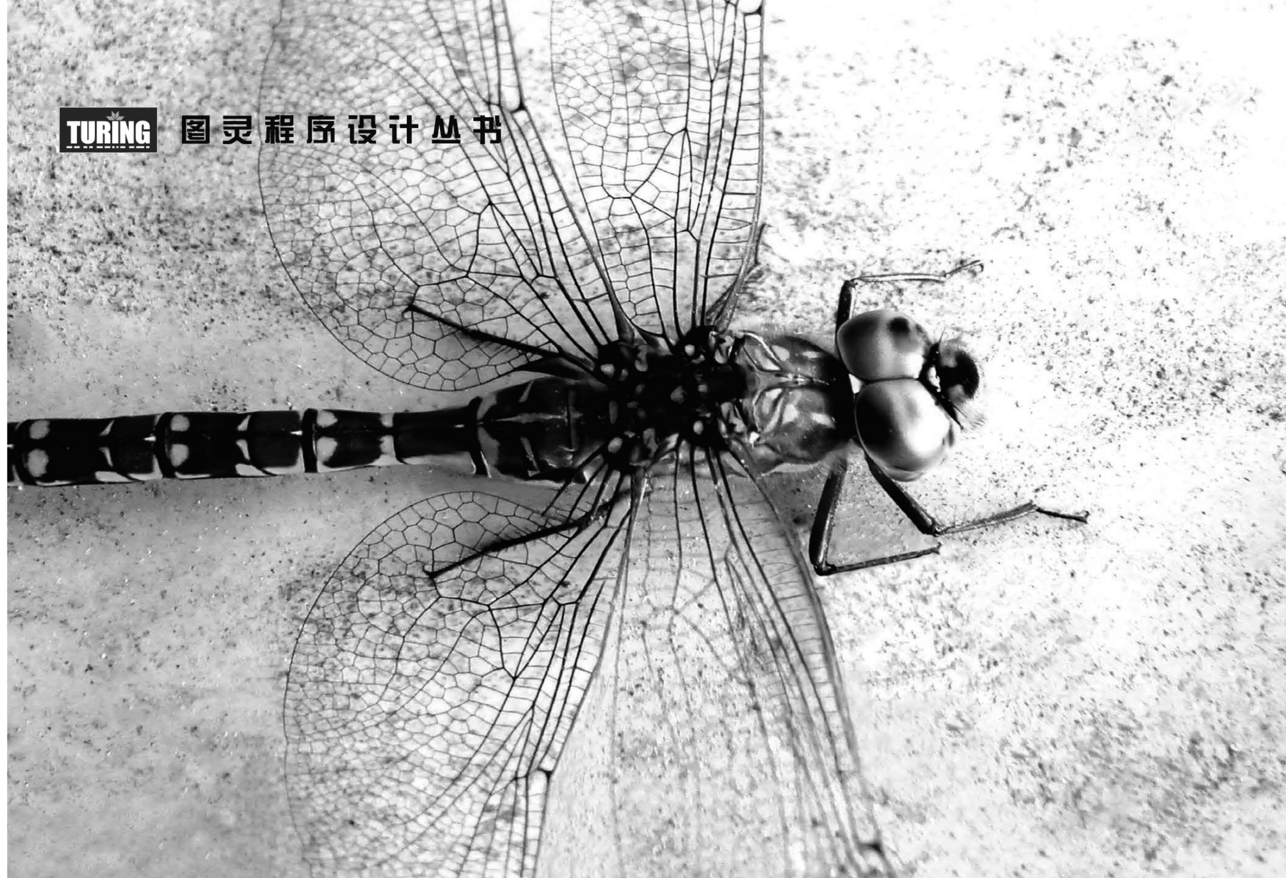
Robert Laganière

渥太华大学电气工程与计算机科学学院教授，同时任教于学院成立的VIVA实验室（主要研究图像与视频处理、计算机视觉、自治系统），还是基于云的家庭监控服务公司iWatchLife和嵌入式视觉解决方案行业引领者Cognivue公司的首席科学家。他与人共同发表过多篇科学论文，并获得了基于内容的视频分析、视觉监控、目标识别和三维重建等领域的多项专利。2006年，他在渥太华与人共同创立了从事视频分析的Visual Cortek公司（2009年被iWatchLife收购）。

个人网站：www.laganiere.name。

相银初

1996年毕业于复旦大学，长期从事软件开发和项目管理工作，涉及C++、C#、Oracle、Linux等技术，也从事软件类图书的翻译工作。



图灵程序设计丛书

[加] Robert Laganière 著　相银初 译

OpenCV 计算机视觉编程攻略（第2版）

OpenCV Computer Vision Application Programming Cookbook, Second Edition

人民邮电出版社
北　京

图书在版编目（CIP）数据

OpenCV计算机视觉编程攻略 ： 第2版 / (加) 拉戈尼尔著 ； 相银初译. -- 北京 : 人民邮电出版社, 2015.9
（图灵程序设计丛书）
ISBN 978-7-115-39850-5

Ⅰ. ①O… Ⅱ. ①拉… ②相… Ⅲ. ①图象处理软件—程序设计 Ⅳ. ①TP391.41

中国版本图书馆CIP数据核字(2015)第156374号

内 容 提 要

本书结合 C++ 和 OpenCV 全面讲解计算机视觉编程，不仅涵盖计算机视觉和图像处理的基础知识，而且通过完整示例讲解OpenCV的重要类和函数。主要内容包括OpenCV库的安装和部署、图像增强、像素操作、图形分析等各种技术，并且详细介绍了如何处理来自文件或摄像机的视频，以及如何检测和跟踪移动对象。

本书适合计算机视觉新手、专业软件开发人员、学生，以及所有想要了解图像处理和计算机视觉技术的人员学习参考。

◆ 著　[加] Robert Laganière
译　相银初
责任编辑　李松峰　毛倩倩
执行编辑　夏长青
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：17　彩插：2
字数：429千字　2015年9月第1版
印数：1－4 000册　2015年9月北京第1次印刷

著作权合同登记号　图字：01-2014-7059号

定价：59.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

译 者 序

计算机视觉：电脑和智能手机的眼睛

人们经常将电脑与人脑作对比。电脑能够处理信息，并且速度非常快，但是电脑和人脑仍有很多本质上的差别，其中之一就是获取信息的方式。人类的信息超过80%是通过视觉获得的，而电脑的信息几乎全部是通过键盘鼠标录入的。你也许会说电脑中有很多多媒体信息，如图像、视频等，但严格来说这些还不能算是“信息”，因为电脑只是存储了它们，并不“认识”它们。电脑存储了大量的照片和视频，但不认识照片上的人是谁，不知道照片中的风景是在哪里拍摄的。

经常看到这样的新闻：某地为了破案，组织几十位警察、花几十个小时查看各路口的监控录像，查找嫌疑人的行踪。为什么这么麻烦？就是因为电脑只是录下视频并存储起来，并没有“认出”视频中的人。否则，只要一条数据库查询语句就解决问题了：“select 时间，视频监控点 from 监控记录 where 视频中的人脸 = 嫌疑人的脸 order by 时间。”

为解决这些问题，人们开始发展计算机视觉这门学科。计算机视觉相当于给电脑装上了真正的眼睛，使它能理解所看到的内容。它的应用非常广泛，从扫描二维码、指纹考勤，到人脸识别、车牌识别、基于内容的图像搜索、根据拍摄的景物自动定位、用手势控制游戏机、根据多幅平面图像还原3D现场、眼球活动操作电脑（科学家霍金）、无人驾驶汽车，等等。

本书特色

OpenCV是计算机视觉领域使用最广泛的开源程序库。本书并不是简单地列出各种函数和类，而是由浅入深地介绍OpenCV及有关算法，让读者从零开始学习计算机视觉和OpenCV，真正掌握相关程序的开发方法。

通过阅读本书，你将了解计算机视觉的基础知识，知道有关算法的来龙去脉，学会OpenCV的总体架构和常用功能，掌握用OpenCV解决具体问题的方法。本书将带你进入图像和视频分析的世界，揭开图像识别、图像配准、视觉跟踪、三维重建等技术的神秘面纱。

翻译过程中的一些体会

说实话，本书的翻译任务还是比较有挑战性的。这主要是因为和纯粹的软件开发类书籍相比，本书所包含的专业术语比较多，并且很多术语并没有统一和规范的中文译法。部分专业术语有多种中文译法，却没有某一种是权威的并且被大家接受的，有的甚至几乎还没有对应的中文术语。而作为一本正式的出版物，如果书中有太多的中英文混杂，不仅不够严谨，而且会让读者产生视觉疲劳，不利于阅读和沟通。因此我在翻译过程中查阅了大量的资料，尽可能在书中使用规范和权威的中文术语，如果确实没有，就选择较为常用的译法。

不过从另一个角度看，这也正好说明国内该领域的开发和应用还处于起步阶段，有相当大的发展前景。随着计算机视觉和OpenCV方面国内开发人员和中文技术资料的增加，该领域将逐步建成统一和规范的中文术语库，专业术语翻译问题很快会得到解决。

致谢

在本书的翻译过程中，我得到了图灵公司李松峰和毛倩倩老师的支持和帮助，在此表示感谢。由于本人水平有限，书中难免有疏忽和错误，恳请读者朋友们批评指正。

2014年12月 于深圳

前 言

OpenCV（Open source Computer Vision）是一个开源程序库，包含了500多个用于图像和视频分析的优化算法。该程序库建立于1999年，现在在计算机视觉领域的研发人员社区中非常流行，被用作主要开发工具。OpenCV最初由英特尔公司的Gary Bradski带领一个小组开发，其目的是推动计算机视觉的研究，促进基于大量视觉处理、CPU密集型应用程序的开发。在一系列beta版本后，1.0版于2006年发布。第二个重要版本是2009年发布的OpenCV 2，它做了一些重要改动，特别是本书所用的新C++接口。OpenCV于2012年改组为一个非营利基金会（http://opencv.org/），依靠众筹进行后续开发。

本书这一版对旧版本中的所有编程方法重新审核和更新，还增加了很多新内容以更全面地覆盖程序库的主要功能点。本书介绍了程序库的很多功能，并且讲述如何使用这些功能完成特定的任务，这样做的目的并不是详细罗列OpenCV中的所有函数和类，而是为读者提供从零起步开发应用的方法。本书还探讨了图像分析的基本概念，介绍了计算机视觉的一些重要算法。

本书将带你进入图像和视频分析的世界，但这只是个开始，因为OpenCV还在不断地演变和扩展。你可以访问OpenCV的在线文档（http://opencv.org/）获得最新资料，也可以访问本书作者的个人网站www.laganiere.name了解有关本书的最新信息。

内容速览

第1章介绍OpenCV库，并演示如何构建一个可以读取并显示图像的简单应用，同时介绍基本的OpenCV数据结构。

第2章解释读取图像的过程，描述了扫描图像的不同方法，让你能在每一个像素上执行操作。

第3章涵盖了各种面向对象设计模式的使用案例，这些设计模式能帮助你更好地构建计算机视觉程序。这一章也讨论了图像中有关颜色的概念。

第4章展示如何计算图像的直方图，以及如何用直方图修改图像。这一章介绍了基于直方图的各种应用，包括图像分割、目标检测和图像检索。

第5章探讨数学形态学的概念，展示不同的算子，并解释如何用这些算子检测图像中的边界、

角点和区段。

第6章讲解频率分析和图像滤波的原理，介绍低通滤波器和高通滤波器在图像处理中的应用，而且介绍了导数算子的概念。

第7章重点介绍几何图像特征的检测方法，解释如何提取图像中的轮廓、直线和连通区域。

第8章介绍图像的几种特征点检测器。

第9章解释如何计算兴趣点描述子，并用其在图像之间匹配兴趣点。

第10章探讨同一场景中两个图像之间的投影关系，同时描述了摄像机校准的过程，且重新探讨匹配特征点的问题。

第11章提出一个读写视频序列和处理帧的框架，同时说明它怎样才能逐帧跟踪特征点，以及如何提取在摄像机前移动的前景物体。

阅读须知

本书基于OpenCV库的C++ API展开介绍，因此你需要有使用C++语言的经验。另外，你还需要有一个良好的C++开发环境以便运行和试用书中的例子；常用的开发环境有Microsoft Visual Studio和Qt。

读者对象

本书适合准备用OpenCV库开发计算机视觉应用的C++初学者，也适合想了解计算机视觉编程概念的专业软件开发人员参考阅读。本书可作为大学计算机视觉课程的学生用书，也是一本非常优秀的参考书，可供图像处理和计算机视觉方面的研究生和科研人员使用。

排版规范

本书使用不同的文本样式区分不同类型的内容，下面是一些样式示例和相关说明。

正文中的代码、用户输入等以这种格式显示：“`cv::Mat`的`create`方法内部封装了这个检查过程，用起来很方便。”

代码块的格式如下：

```
// 用Mat_模板操作图像
cv::Mat_<uchar> im2(image);
 im2(50,100)= 0; // 访问第50行、第100列处那个值
```

读者反馈

我们一贯欢迎读者的反馈意见。请告诉我们你对本书的看法——喜欢或不喜欢哪些内容。读者的反馈对于协助我们创作出真正对读者有所裨益的内容至关重要。

一般性的反馈意见，请直接发邮件到feedback@packtpub.com，并在邮件标题中注明书名。

如果你是某一方面的专家并愿意参与写作或合作著书，请访问www.packtpub.com/authors查看作者指南。

客户支持

现在你已经拥有了一本由Packt出版的书，为了让你的付出得到最大的回报，我们还为你提供了其他许多方面的服务，请注意以下信息。

下载代码

如果你是通过http://www.packtpub.com上的注册账户购买的图书，可以从该账户下载相应Packt图书的示例代码[①]。如果你是从其他地方购买的本书英文版，那么可以访问http://www.acktpub.com/support并注册，然后会通过邮件接收到文件。

勘误

我们已经尽最大努力确保内容准确，但错误仍在所难免。如果你发现书中有错（文字或代码错误），请告诉我们，这可让其他读者免于困惑，也可帮助我们在后续版本中加以改进。发现错误后，请访问http://www.packtpub.com/submit-errata，选择对应的图书，点击链接errata submission form（提交勘误表[②]）登记错误详情。勘误通过核实后，你提交的错误信息会上传到网站或添加到该书已有的勘误表中。你可以在http://www.packtpub.com/support中通过书名查看已有的勘误表。

举报盗版

网络盗版是个老问题了。在Packt，我们非常重视版权和许可。如果你在网上见到对我们作品的任何形式的非法复制品，请将网址或网站名称及时告知我们，以便我们采取补救措施。

① 本书中文版的读者可免费注册iTuring.cn，至本书页面下载。——编者注
② 中文版的勘误请注册iTuring.cn，至本书页面提交。——编者注

请将疑似盗版内容的链接发送到copyright@packtpub.com。

非常感谢你的帮助，这不仅将保护作者权益，也让我们有能力为大家提供有价值的内容。

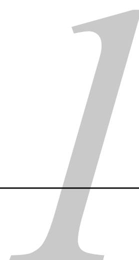
疑难解答

如果你有关于本书的任何疑问，请通过questions@packtpub.com联系我们，我们会尽力解决。

目　　录

第 1 章

图像编程入门



本章我们开始学习OpenCV库。你将学习：

- 安装OpenCV库；
- 装载、显示和保存图像；
- 深入理解cv::Mat数据结构；
- 定义ROI区域（感兴趣区域）。

1.1　简介

本章介绍OpenCV的基本要素，并演示如何完成最基本的图像处理任务：读取、显示和保存图像。开始之前，首先需要安装OpenCV库。安装过程非常简单，1.2节会详细介绍。

所有的计算机视觉应用程序都包含对图像的处理。因此，OpenCV提供的最基础的工具即为一个操作图像和矩阵的数据结构。此数据结构功能非常强大，具有多种实用属性和方法。此外，它还包含先进的内存管理模型，十分有助于应用程序的开发。本章最后两节介绍如何使用这个重要的OpenCV数据结构。

1.2　安装 OpenCV 库

OpenCV是一个开源的程序库，用于开发在Windows、Linux、Android和Mac OS系统中运行的计算机视觉应用程序。在BSD许可协议下，它可以用来开发学术应用和商业应用，可随意使用、发布和修改。本节介绍如何安装OpenCV程序库。

1.2.1　准备工作

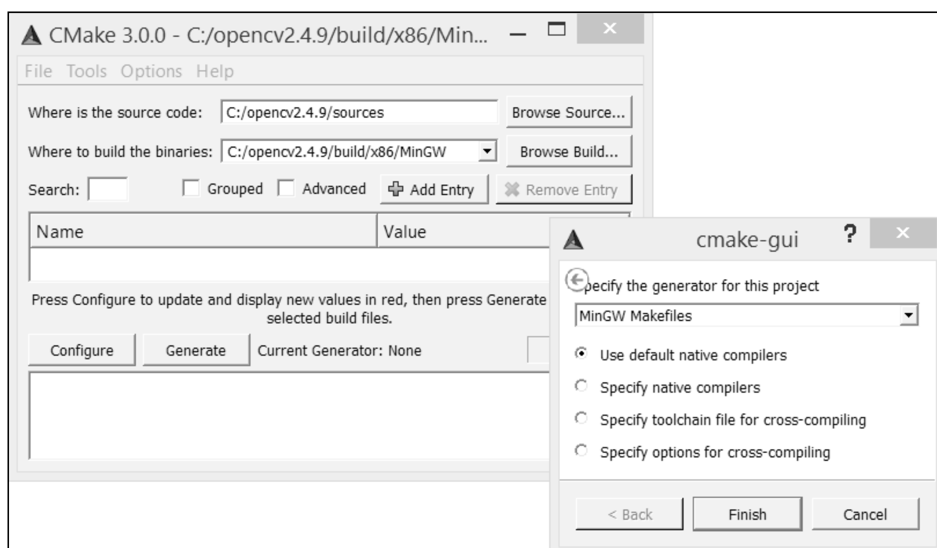
访问OpenCV官方网站http://opencv.org/，你可以看到最新版程序库、在线文档以及诸多其他有用的资源。

1.2.2　安装

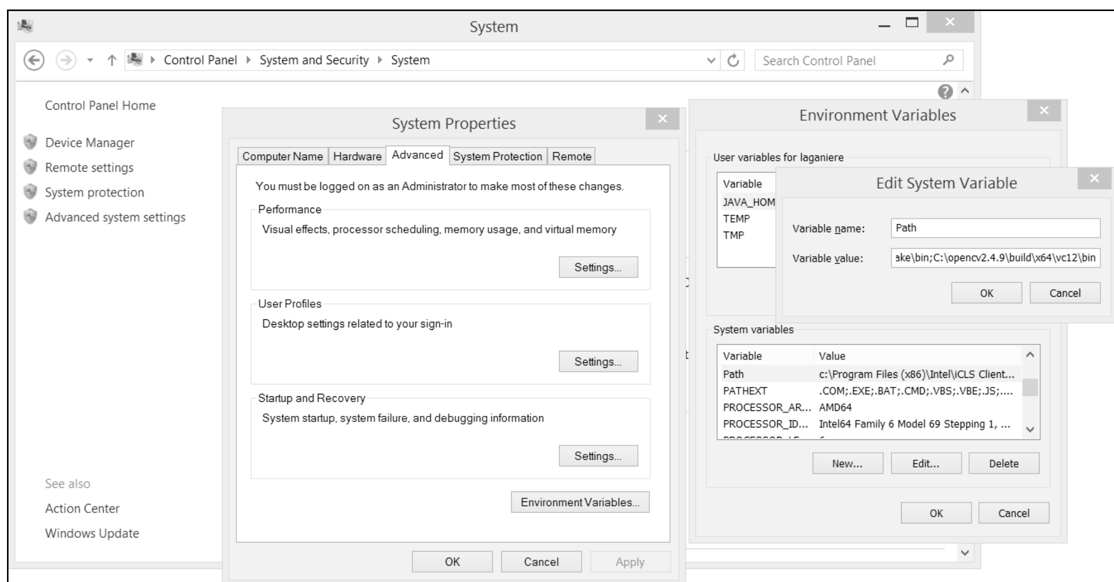
在OpenCV网站上选择你使用的平台（Unix、Windows或Android），并转到相应的**DOWNLOADS**（下载）页面，从那里下载OpenCV包，然后将其解压；解压的目录名通常与程序库版本一致（例如，在Windows下可解压到C:\OpenCV2.4.9）。解压之后，你会看到很多构成程序库的文件和目录。注意有一个sources目录，它包含所有的源代码文件。（是的，它是开源的！）但是，要完成程序库安装并使之能使用，你还需执行一步操作：针对所选环境生成程序库的二进制文件。这时，你必须选定创建OpenCV程序所用的目标平台。使用哪种操作系统？Windows还是Linux？使用什么编译器？Microsoft VS2013还是MinGW？32位还是64位？你在开发项目时将要使用的集成开发环境（IDE）也会引导你做这些选择。

注意，如果你在装有Visual Studio的Windows环境下操作，可执行安装包很可能不仅仅安装源文件，还可能安装构建应用程序所需的已编译的二进制文件。检查一下build目录，它应该包含子目录x64和x86（分别对应64位和32位版本）。这些子目录下有vc10、vc11、vc12等目录，这些目录包含了用于不同版本MS Visual Studio的二进制文件。在此情况下，除非你想使用特殊的选项进行个性化构建，否则可以跳过本节讲解的编译过程直接使用OpenCV。

为了完成安装过程并构建OpenCV二进制文件，你需要使用**CMake**工具，该工具可从http://cmake.org下载。CMake是另一个开源软件工具，用于控制使用了跨平台配置文件的软件系统的编译过程。它可以生成在特定环境下编译软件库所需的**makefile**和**workspace**文件。因此你需要事先下载并安装CMake，之后可以使用命令行工具来运行CMake，但更容易的方法是使用GUI工具（`cmake-gui`）。使用GUI，你只需要指定OpenCV库源文件和二进制文件所在的目录，点击**Configure**按钮来选择适合的编译器，然后再一次点击**Configure**按钮。



现在可以点击**Generate**按钮生成项目文件了，这些项目文件用来编译程序库。这是安装过程的最后一个步骤，会生成能在指定的开发环境下使用的程序库。例如，如果你选用Visual Studio，那么只需要打开由CMake创建的位于顶层的解决方案文件（通常是OpenCV.sln文件），然后在Visual Studio中输入`Build Solution`命令。如果要得到`Release`和`Debug`两个版本，你就需要编译两次——每个相应的配置一次。在已创建的目录bin下包含动态库文件，可执行文件在运行时需要调用这些动态库文件。别忘了在控制面板中设置环境变量`PATH`，以确保运行程序时操作系统能找到这些dll文件。



在Linux环境中，你可用`make`实用命令运行前面生成的makefile文件。为了完成所有目录的安装，你需要运行`Build INSTALL`或者`sudo make INSTALL`命令。

在构建程序库之前，一定要检查一下OpenCV安装程序产生的内容；安装程序可能已经产生了程序库，那样就可免去编译的步骤。如果要使用Qt作为IDE，1.2.4节描述了编译OpenCV项目的另一种方法。

1.2.3 实现原理

从2.2版开始，OpenCV库就分成了几个模块。这些模块是内置的库文件，位于lib目录下。其中常用的模块有：

- `opencv_core`模块，包含了程序库的核心功能，特别是基本的数据结构和算法函数；
- `opencv_imgproc`模块，包含了主要的图像处理函数；

- opencv_highgui模块，包含图像、视频读写函数和部分用户界面函数；
- opencv_features2d模块，包含特征点检测器、描述子以及特征点匹配框架；
- opencv_calib3d模块，包含相机标定、双视角几何估计以及立体函数；
- opencv_video模块，包含运动估计、特征跟踪以及前景提取函数和类；
- opencv_objdetect模块，包含目标检测函数，例如面部和人体探测器。

OpenCV库还包含了其他实用模块：机器学习函数（opencv_ml）、计算几何算法（opencv_flann）、共享代码（opencv_contrib）、过时的代码（opencv_legacy）以及GPU加速代码（opencv_gpu）。此外还有一些专门用来实现较高层次函数的库，例如用于计算摄影的opencv_photo和实现图像拼接算法的opencv_stitching。另外还有一个opencv_nonfree模块，它包含在使用过程中可能有限制的函数。如果程序用到了某些OpenCV函数，你就必须在编译时将程序与包含这些函数的库链接。一般来说，使用刚才列出的前三个模块，然后根据具体程序的作用域选择其他模块。

所有这些模块都有一个对应的头文件（位于include目录中）。因此，典型的OpenCV C++代码会首先包含必需的模块。例如（这是推荐的声明样式）：

```
#include <opencv2/core/core.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
```

下载示例代码



若用Packt账号购买了本书英文版，你可以从http://www.packtpub.com下载示例代码文件[①]。如果你是从其他地方购买的本书英文版，那么可以访问http://www.packtpub.com/support并注册，然后会通过邮件接收到文件。

你可能会看到用以下命令开头的OpenCV代码：

```
#include "cv.h"
```

这是因为在程序库被重构成多个模块之前，应用使用了老式的风格。最后要注意的是，以后OpenCV还会被重构；因此，如果你下载2.4以后的版本，模块的划分可能会不同。

1.2.4　扩展阅读

OpenCV网站http://opencv.org/上有详细的安装说明，还有完整的在线文档，包括几个针对程序库中不同组件的教程。

---

① 本书中文版的读者可免费注册iTuring.cn，至本书页面下载。——编者注

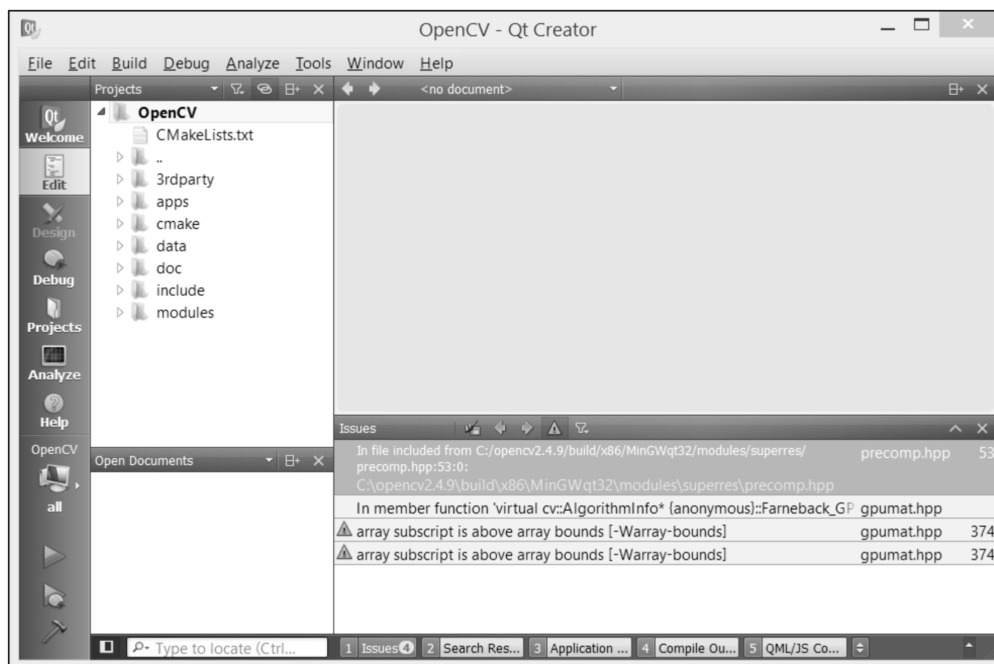
1. 使用Qt进行OpenCV开发

Qt是开发C++应用程序的跨平台IDE，是作为开源项目发展起来的。你可以在LPGL开源协议下使用Qt，也可以在商业（付费）协议下用Qt开发专有项目。它由两个独立的部分组成：一个称为Qt creator的跨平台IDE、一系列Qt类库和开发工具。使用Qt来开发C++应用程序有以下好处：

- ❑ 它是由Qt社区发起的开源项目，提供各种Qt组件的源代码；
- ❑ 它是一个跨平台IDE，这意味着开发的应用程序能在不同操作系统上运行，如Windows、Linux、Mac OS X等；
- ❑ 它包含了一个完整并且跨平台的GUI库，遵循高效的面向对象和事件驱动的模型；
- ❑ Qt还包含几个跨平台库，有助于开发多媒体、图形、数据库、多线程、Web应用以及很多用于设计高级应用程序的有趣的构建模块。

你可以从http://qt-project.org/下载Qt。在安装Qt时需要选择不同的编译器。在Windows下，MinGW是一款可以用来取代Visual Studio的非常不错的编译器。

因为Qt可以读取CMake文件，用它来编译OpenCV特别容易。在安装OpenCV和CMake后，你只需要在Qt菜单中选择**Open File**或者**Project...**，打开OpenCV中sources目录下的CMakeLists.txt文件，这样就会生成一个用`Build Project Qt`命令构建的OpenCV项目：



可能会有几个警告信息，但你可以不管它们。

2. OpenCV开发者站点

OpenCV是一个开源项目，非常欢迎用户做出贡献。你可以访问开发者网站http://code.opencv.org。除此之外，你也可以获得当前已经开发完毕的OpenCV版本。这个社区使用Git作为版本控制系统，因此必须使用Git来获得最新版本的OpenCV。作为一个免费、开源的软件系统，Git可能是管理源代码的最好工具，它可在http://git-scm.com/下载。

1.2.5　参阅

- 作者的网站（www.laganiere.name）上有安装最新版本OpenCV库的详细步骤；
- 1.3.4节详解如何用Qt创建OpenCV项目。

1.3　装载、显示和存储图像

现在我们开始运行第一个OpenCV应用程序。既然OpenCV是处理图像的，这里我们来演示几个图像程序开发中最基本的操作，即从文件中装载一个输入的图像、在窗口中显示图像、应用一个处理函数，然后把输出图像存储到磁盘。

1.3.1　准备工作

使用你喜欢的IDE（例如MS Visual Studio或者Qt）新建一个控制台应用程序，使用待填充内容的`main`函数。

1.3.2　如何实现

首先要引入头文件，这些头文件定义了所需的类和函数。这里我们只是简单地显示一个图像，因此需要定义了图像数据结构的核心库和包含了所有图形接口函数的highgui头文件：

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
```

在`main`函数中，首先定义一个表示图像的变量。在OpenCV 2中，定义`cv::Mat`类的对象：

```
cv::Mat image; // 创建一个空图像
```

这个定义创建了一个尺寸为`0×0`的图像。可以访问`cv::Mat`的`size`属性来验证这一点：

```
std::cout << "This image is " << image.rows << " x "
          << image.cols << std::endl;
```

接下来只需调用读函数，即会读入一个图像文件，解码，然后分配内存：

```
image= cv::imread("puppy.bmp"); // 读取输入图像
```

现在可以使用这个图像了。但是要先检查图像的读取是否正确（如果找不到文件、文件被破坏或者文件格式无法识别，就会发生错误）。用下面的代码来验证图像是否有效：

```
if (image.empty()) { // 错误处理
    // 未创建图像……
    // 可能显示一个错误消息
    // 并退出程序
    ...
}
```

如果没有分配图像数据，empty方法返回true。

对这个图像的第一个操作就是显示它。你可以使用highgui模块的函数来实现，先定义用来显示图像的窗口，然后让图像在指定的窗口中显示：

```
// 定义窗口（可选）
cv::namedWindow("Original Image");
// 显示图像
cv::imshow("Original Image", image);
```

可以看到，这个窗口是用名称来标识的。我们稍后可以重用这个窗口来显示其他图像，也可以用不同的名称创建多个窗口。运行这个应用程序，可看到如下的图像窗口：



这时，我们通常会对图像做一些处理。OpenCV提供了众多的处理函数，本书将对其中一些进行深入探讨。我们先来看一个将图像水平翻转的简单函数。OpenCV中有些图像转换过程是就地进行的，即转换过程直接在输入的图像上进行（不创建新的图像）。这里讲的是用翻转方法转化图像的例子。不过，我们总是可以创建一个新的矩阵来存放输出结果，下面就采用这种方法：

```
cv::Mat result; // 创建另一个空的图像
cv::flip(image,result,1); // 正数表示水平，
                          // 0表示垂直，
                          // 负数表示水平和垂直
```

在另一个窗口显示结果：

```
cv::namedWindow("Output Image"); // 输出窗口
cv::imshow("Output Image", result);
```

因为它是控制台窗口，会在main函数结束时关闭，所以我们增加一个额外的highgui函数，需要用户键入数值才能结束程序：

```
cv::waitKey(0); // 0表示永远地等待按键，
                // 正数表示等待指定的毫秒数
```

我们可以在另一个窗口上看到输出的图像，如下：



最后可以使用下面的highgui函数把处理过的图像存储在磁盘里：

```
cv::imwrite("output.bmp", result); // 保存结果
```

保存图像时会根据文件名后缀决定使用哪种编码方式。其他常见的受支持图像格式是JPG、TIFF和PNG。

1.3.3　实现原理

在OpenCV的C++ API中，所有类和函数都在命名空间cv内定义。我们有两种方法可以访问它们，第一种方法是在定义main函数前使用如下声明：

```
using namespace cv;
```

第二种方法是使用命名空间规范给所有OpenCV的类和函数加上前缀cv::，本书即采用这种方法。使用前缀可让OpenCV的类和函数更容易识别。

在highgui模块中有一批函数可用来方便地显示图像和对图像进行操作。在使用imread函数装载图像时，你可以通过设置选项把它转换为灰度图像。这个选项非常实用，因为有些计算机

视觉算法是必须使用灰度图像的。在读入图像的同时进行色彩转换，这可以提高运行速度并减少内存使用。做法如下：

```
// 读入一个图像文件并转换为灰度图像
image= cv::imread("puppy.bmp", CV_LOAD_IMAGE_GRAYSCALE);
```

这样生成的图像由无符号字节（C++中为unsigned char）构成，OpenCV中用定义的常量CV_8U表示。另外，即使图像是作为灰度图像保存的，有时仍需要在读入时把它转换成三通道彩色图像。要实现这个功能，可把imread函数的第二个参数设置为正数：

```
// 读取图像，并转换为三通道彩色图像
image= cv::imread("puppy.bmp", CV_LOAD_IMAGE_COLOR);
```

这样创建的图像中每个像素有3字节，OpenCV中用CV_8UC3表示。当然了，如果输入的图像文件是灰度图像，这三个通道的值就是相同的。最后，如果要在读入图像时采用文件本身的格式，只需把第二个参数设置为负数。可用channels方法检查图像的通道数：

```
std::cout << "This image has "
          << image.channels() << " channel(s)";
```

注意，当用imread打开路径指定不完整的图像时（前面例子的做法），imread会自动采用默认目录。如果从控制台运行程序，默认目录显然就是可执行文件所在的目录。但是如果直接在IDE中运行程序，这个默认目录通常就是项目文件所在的目录，因此要确保图像文件在正确的目录下。

如果用imshow显示的图像是由整数（CV_16U表示16位无符号整数，CV_32S表示32位有符号整数）构成的，图像每个像素的值会被除以256，以便能够在256级灰度中显示。同样，在显示由浮点数构成的图像时，值的范围会被假设为从0.0（显示黑色）到1.0（显示白色）。超出这个范围的值会显示为白色（大于1.0的值）或黑色（小于0.0的值）。

highgui模块非常适用于构建原型程序。在生成最终版本的程序时，你很可能会用到IDE提供的GUI模块，这样会让程序看起来更专业。

这个程序同时使用了输入图像和输出图像，练习时你可以对这个简单程序做一下改动，改成就地处理的方式，也就是不定义输出图像而直接写入原图像：

```
cv::flip(image,image,1); // 就地处理
```

1.3.4　扩展阅读

highgui模块中有大量可用来处理图像的函数，运用这些函数可以使程序对鼠标或键盘事件做出响应，也可以在图像上绘制轮廓或写入文本。

1. 在图像上点击

你可以通过编程实现鼠标置于图像窗口上时运行特定的指令。要实现这个功能，需定义一个

适当的回调函数。回调函数不会被显式地调用，但是会在响应特定事件（这里是指有关鼠标与图像窗口交互的事件）的时候被程序调用。为了能被程序识别，回调函数需要具有特定的签名，并且必须注册。对于这种鼠标事件处理函数，回调函数必须具有这种签名：

```
void onMouse( int event, int x, int y, int flags, void* param);
```

第一个参数是整数，表示触发回调函数的鼠标事件的类型。后面两个参数是事件发生时鼠标的位置，用像素坐标表示。参数flags表示事件发生时按下了鼠标的哪个键。最后一个参数是执行任意对象的指针，作为附加的参数发送给函数。你可用下面的方法在程序中注册回调函数：

```
cv::setMouseCallback("Original Image", onMouse,
                     reinterpret_cast<void*>(&image));
```

本例中，函数onMouse与名为**Original Image**（原始图像）的图像窗口建立关联，同时把所显示图像的地址作为附加参数传给函数。现在，只要用下面的代码定义回调函数onMouse，每当遇到鼠标点击事件时，控制台中就会显示对应像素的值（这里我们假定它是灰度图像）。

```
void onMouse( int event, int x, int y, int flags, void* param) {

  cv::Mat *im= reinterpret_cast<cv::Mat*>(param);

  switch (event) { // 调度事件

    case CV_EVENT_LBUTTONDOWN: // 鼠标左键按下事件

      // 显示像素值(x,y)
      std::cout << "at (" << x << "," << y << ") value is: "
        << static_cast<int>(
                  im->at<uchar>(cv::Point(x,y))) << std::endl;
      break;
  }
}
```

这里用cv::Mat对象的at方法来获取（x, y）的像素值，第2章会详细讨论这个方法。鼠标事件的回调函数可能收到的事件还有：CV_EVENT_MOUSEMOVE、CV_EVENT_LBUTTONUP、CV_EVENT_RBUTTONDOWN、CV_EVENT_RBUTTONUP。

2. 在图像上绘图

OpenCV还提供了几个用于在图像上绘制形状和写入文本的函数。基本的形状绘制函数有circle、ellipse、line、rectangle。这是一个使用circle函数的例子：

```
cv::circle(image,              // 目标图像
        cv::Point(155,110),    // 中心点坐标
        65,                    // 半径
        0,                     // 颜色（这里用黑色）
        3);                    // 厚度
```

在OpenCV的方法和函数中，我们经常用`cv::Point`结构来表示像素的坐标。这里假定是在灰度图像是进行绘制，因此我们用单个整数来表示颜色。在1.4节我们将学习如何使用`cv::Scalar`结构表示彩色图像颜色值。你也可以在图像上写入文本，方法如下：

```
cv::putText(image,                  // 目标图像
        "This is a dog.",           // 文本
        cv::Point(40,200),          // 文本位置
        cv::FONT_HERSHEY_PLAIN,     // 字体类型
        2.0,                        // 字体大小
        255,                        // 字体颜色（这里用白色）
        2);                         // 文本厚度
```

在测试图像上调用上述两个函数后，得到的结果如下图所示：



3. 用Qt运行示例程序

要使用Qt运行OpenCV应用程序，你需要创建项目文件。针对本节中的例子，项目文件（loadDisplaySave.pro）显示如下：

```
QT        += core
QT        -= gui

TARGET = loadDisplaySave
CONFIG    += console
CONFIG    -= app_bundle

TEMPLATE = app

SOURCES += loadDisplaySave.cpp
INCLUDEPATH += C:\OpenCV2.4.9\build\include
LIBS += -LC:\OpenCV2.4.9\build\x86\MinGWqt32\lib \
-lopencv_core249 \
-lopencv_imgproc249 \
-lopencv_highgui249
```

这个项目文件表明了头文件和库文件所在的路径，还列出了示例程序所用的库模块。请确保选用的库文件与Qt编译器兼容。网上下载的本书示例程序的源代码中包含了CMakeLists文件，可以用Qt（或CMake）打开，用来创建有关项目。

1.3.5　参阅

- ❑ `cv::Mat`类是用来存放图像（以及其他矩阵数据）的数据结构。在所有OpenCV类和函数中，这个数据结构具有核心地位，1.4节我们将对它做详细介绍。
- ❑ 你可以从这里下载本书示例程序的源代码：https://github.com/laganiere/。

1.4　深入了解 cv::Mat

1.3节提到了`cv::Mat`数据结构。正如前面所说，它是程序库中的关键元素，用来操作图像和矩阵（从计算机和数学的角度看，图像其实就是矩阵）。在开发程序时你会经常用到这个数据结构，因此有必要熟悉它。通过本节的学习你将了解到它采用了很巧妙的内存管理机制，因此支持高效的内存使用。

1.4.1　如何实现

下面的程序可用来测试`cv::Mat`数据结构的不同属性：

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>

// 测试函数，它创建一个图像
cv::Mat function() {
   // 创建图像
   cv::Mat ima(500,500,CV_8U,50);
   // 返回图像
   return ima;
}

int main() {
  // 定义图像窗口
  cv::namedWindow("Image 1");
  cv::namedWindow("Image 2");
  cv::namedWindow("Image 3");
  cv::namedWindow("Image 4");
  cv::namedWindow("Image 5");
  cv::namedWindow("Image");

  // 创建一个240行 × 320列的新图像
  cv::Mat image1(240,320,CV_8U,100);
```

```
cv::imshow("Image", image1); // 显示图像
cv::waitKey(0); // 等待按键

// 重新分配一个新的图像
image1.create(200,200,CV_8U);
image1= 200;

cv::imshow("Image", image1); // 显示图像
cv::waitKey(0); // 等待按键

// 创建一个红色的图像
// 通道次序为BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));

// 或者：
// cv::Mat image2(cv::Size(320,240),CV_8UC3);
// image2= cv::Scalar(0,0,255);

cv::imshow("Image", image2); // 显示图像
cv::waitKey(0); // 等待按键

// 读入一个图像
cv::Mat image3= cv::imread("puppy.bmp");

// 所有这些图像都指向同一个数据块
cv::Mat image4(image3);
image1= image3;

// 这些图像是源图像的副本图像
image3.copyTo(image2);
cv::Mat image5= image3.clone();

// 转换图像用来测试
cv::flip(image3,image3,1);

// 检查哪些图像在处理过程中受到了影响
cv::imshow("Image 3", image3);
cv::imshow("Image 1", image1);
cv::imshow("Image 2", image2);
cv::imshow("Image 4", image4);
cv::imshow("Image 5", image5);
cv::waitKey(0); // 等待按键


// 从函数中获取一个灰度图像
cv::Mat gray= function();

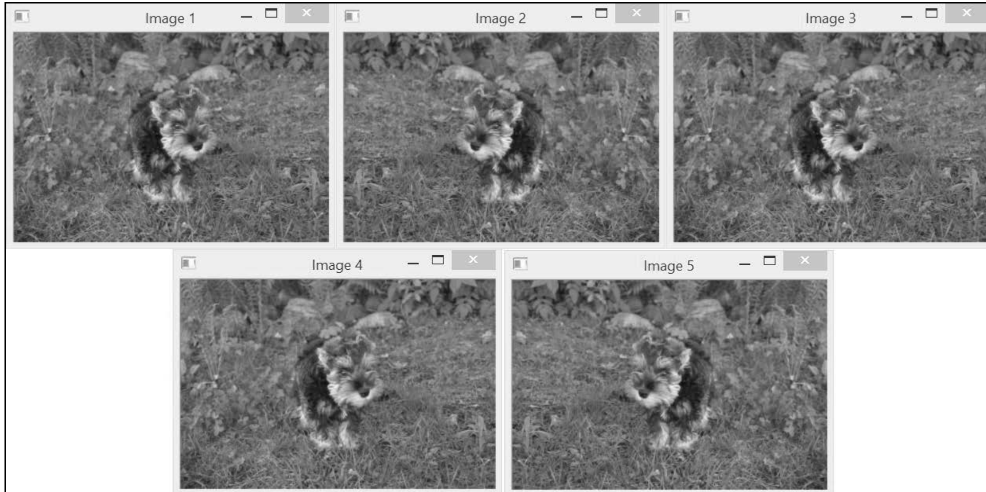
cv::imshow("Image", gray); // 显示图像
cv::waitKey(0); // 等待按键

// 作为灰度图像读入
image1= cv::imread("puppy.bmp", CV_LOAD_IMAGE_GRAYSCALE);
image1.convertTo(image2,CV_32F,1/255.0,0.0);
```

```
  cv::imshow("Image", image2); // 显示图像
  cv::waitKey(0); // 等待按键

  return 0;
}
```

运行这个程序，你将得到下面这些图像：



1.4.2　实现原理

cv::Mat 有两个必不可少的组成部分：一个头部和一个数据块。头部包含了矩阵的所有相关信息（大小、通道数量、数据类型等），1.3节介绍了如何访问cv::Mat头部文件的某些属性（例如，通过使用cols、rows或channels）。数据块包含了图像中所有像素的值。头部有一个指向数据块的指针，即data属性。cv::Mat有一个很重要的属性，即只有在明确要求时，内存块才会被复制。实际上，大多数操作仅仅复制了cv::Mat的头部，因此多个对象会同时指向同一个数据块。这种内存管理模式可以提高应用程序的运行效率，避免内存泄漏，但是我们必须了解它带来的后果。本节的例子会对这点进行说明。

新创建的cv::Mat对象默认大小为0，但也可以指定一个初始大小，例如：

```
// 创建一个240行 × 320列的新图像
cv::Mat image1(240,320,CV_8U,100);
```

我们需要指定每个矩阵元素的类型，这里我们用CV_8U表示每个像素对应1字节，用字母U表示无符号；你也可用字母S表示有符号。对于彩色图像，你应该用三通道类型（CV_8UC3），也可以定义16位和32位的整数（有符号或无符号），例如CV_16SC3。我们甚至可以使用32位和64位的浮点数（例如CV_32F）。

图像（或矩阵）的每个元素都可以包含多个值（例如彩色图像中的三个通道），因此OpenCV引入了一个简单的数据结构cv::Scalar，用于在调用函数时传递像素值。该结构通常包含一个或三个值。例如，要创建一个彩色图像并用红色像素初始化，可用如下代码：

```
// 创建一个红色图像
// 通道次序是BGR
cv::Mat image2(240,320,CV_8UC3,cv::Scalar(0,0,255));
```

类似地，初始化灰度图像可这样使用这个数据结构：cv::Scalar(100)。

图像的大小信息通常也需要传递给调用函数。前面讲过，我们可以用属性cols和rows来获得cv::Mat实例的大小。cv::Size结构包含了矩阵高度和宽度，同样可以提供图像的大小信息。另外，我们可用size()方法来得到当前矩阵的大小。当需要指明矩阵的大小时，很多方法都使用这种格式。

例如，可以这样创建一个图像：

```
// 创建一个未初始化的彩色图像
cv::Mat image2(cv::Size(320,240),CV_8UC3);
```

我们可以随时用create方法分配或重新分配图像的数据块，如果图像已经分配，首先其原来的内容会被释放。处于对性能上的考虑，如果新的大小和类型与原来的相同，就不会重新分配内存：

```
// 重新分配一个新图像
// （仅在大小或类型不同时）
image1.create(200,200,CV_8U);
```

一旦没有了指向cv::Mat对象的引用，分配的内存就会被自动释放。这一点非常方便，因为它避免了C++动态内存分配中经常发生的内存泄漏问题。这是OpenCV 2中一个关键的机制，它的实现方法是通过cv::Mat实现计数引用和浅复制。因此，当在两个图像之间赋值时，图像数据（即像素）并不会被复制，此时两个图像都指向同一个内存块。这同样适用于图像间的值传递或值返回。由于维护了一个引用计数器，因此只有当图像的所有引用都将释放或赋值给另一个图像时，内存才会被释放；

```
// 所有图像都指向同一个数据块
cv::Mat image4(image3);
image1= image3;
```

上面的图像中，对其中的任何一个做转换都会影响到其他图像。如果要对图像内容做一个深复制，你可以使用copyTo方法，在此情况下目标图像会调用create方法。另一个生成图像副本的方法是clone，即创建一个完全相同的新图像：

```
// 这些图像是原始图像的新副本
image3.copyTo(image2);
cv::Mat image5= image3.clone();
```

如果你需要把一个图像复制到另一个图像中，而两者的数据类型不一定相同，那就要使用convertTo方法：

```
// 转换成浮点型图像 [0,1]
image1.convertTo(image2,CV_32F,1/255.0,0.0);
```

本例中，原始图像被复制进了一个浮点型图像。这一方法包含两个可选参数：缩放比例和偏移量。需要注意的是，这两个图像的通道数量必须相同。

cv::Mat 对象的分配模型还能让程序员安全地编写返回一个图像的函数（或类方法）：

```
cv::Mat function() {

   // 创建图像
   cv::Mat ima(240,320,CV_8U,cv::Scalar(100));
   // 返回图像
   return ima;
}
```

我们还可以从main函数中调用这个函数：

```
// 得到一个灰度图像
cv::Mat gray= function();
```

运行这条语句后，我们就可以用变量gray操作这个由function函数创建的图像，而不需要额外分配内存。正如前面解释的，从cv::Mat实例到图像gray，实际上只是进行了一次浅复制。当局部变量ima超出作用范围后，ima会被释放，但是从相关引用计数器可以看出，有另一个实例（即变量gray）引用了ima内部的图像数据，因此ima的内存块不会被释放。

注意，在使用类的时候要特别小心，不要回送图像的类属性。下面的实现方法很容易引发错误：

```
class Test {
   // 图像属性
   cv::Mat ima;
  public:
     // 在构造函数中创建一个灰度图像
     Test() : ima(240,320,CV_8U,cv::Scalar(100)) {}

     // 用这种方法回送一个类属性，这是一种不好的做法
     cv::Mat method() { return ima; }
};
```

这里，如果某个函数调用了这个类的method，就会对图像属性进行一次浅复制。一旦副本稍后被修改了，class属性也会被“偷偷地”修改，这会影响这个类的后续行为（反过来也一样）。为了避免这种类型的错误，你需要将其改成回送属性的一个副本。

1.4.3　扩展阅读

OpenCV中还有几个与cv::Mat相关的类，熟练掌握这些类也很重要。

1. 输入和输出数组

在OpenCV的文档中，有很多方法和函数使用cv::InputArray类型作为输入参数。cv::InputArray类型是一个简单的代理类，用来概括OpenCV中数组的概念，避免同一个方法或函数因为使用了不同类型的输入参数而出现多个不同的版本。也就是说，你可以在参数中使用cv::Mat对象或者其他的兼容类型。cv::InputArray只是一个接口，因此你不能在代码中显式地定义它。比较有趣的是，cv::InputArray也能使用常见的std::vector类来构造，这意味着std::vector的对象可作为内容对象输入OpenCV的算法和函数（只要这么做是有意义的）。其他兼容的类型有**cv::Scalar**和**cv::Vec**，其中**cv::Vec**将在下一章介绍。此外还有一个代理类**cv::OutputArray**，用来指定某些方法或函数的返回数组。

2. 老式的IplImage结构

在OpenCV第2版中引入了一个新的C++接口。早期版本使用C语言风格的函数和结构（现在仍能使用）。特别是用IplImage结构来操作图像，该结构是从**IPL**库继承的（即Intel Image Processing库），现在已经与**IPP**库（即Intel Integrated Performance Primitive库）合并。如果使用老式的C语言接口创建的代码和库，你就需要操作这些IplImage结构。幸运的是，把IplImage结构转换成cv::Mat对象非常容易，如下面的代码所示：

```
IplImage* iplImage = cvLoadImage("puppy.bmp");
cv::Mat image(iplImage,false);
```

cvLoadImage是用C语言接口装载图像的函数。cv::Mat对象的构造函数中，第二个参数表示不复制数据（如需要复制，可把它设为true；默认值是false，因此可以省略），这意味着IplImage和image会共用同一块图像数据。这里你需要特别小心，以避免产生悬挂指针，因此更安全的做法是把IplImage指针封装进OpenCV 2的引用计数指针类中：

```
cv::Ptr<IplImage> iplImage = cvLoadImage("puppy.bmp");
```

否则，当释放IplImage结构指向的内存时需要显式执行：

```
cvReleaseImage(&iplImage);
```

记住，请不要使用这个过时的数据结构，要改用cv::Mat数据结构。

1.4.4 参阅

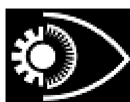
- 要查看完整的OpenCV文档，请访问http://docs.opencv.org/。
- 第2章将介绍如何高效地访问和修改cv::Mat表示的图像的像素值。
- 1.5节将解释如何定义图像内的兴趣区域。

1.5　定义兴趣区域

有时我们需要让一个处理函数只在图像的某个部分起作用。OpenCV内嵌了一个精致又简洁的机制，可以定义图像的子区域，并把这个子区域当作普通图像进行操作。本节介绍如何定义图像内部的兴趣区域。

1.5.1　准备工作

假设我们要把一个小图像复制到一个大图像上。例如，我们要把下面的小标志插入到测试图像中。



为了实现这个功能，我们可以定义一个兴趣区域（Region Of Interest，ROI），在它上面进行复制操作，这个ROI的位置将决定标志的插入位置。

1.5.2　如何实现

第一步是定义ROI。定义后，我们可以把ROI当作一个普通的`cv::Mat`实例进行操作。关键在于，ROI实际上就是一个`cv::Mat`对象，它与它的父图像指向同一个数据缓冲区，并且有一个头部信息表示ROI的坐标。接着我们可以用下面的方法插入小标志：

```
// 在图像的右下角定义一个ROI
cv::Mat imageROI(image,
            cv::Rect(image.cols-logo.cols, // ROI坐标
                     image.rows-logo.rows,
                     logo.cols,logo.rows));// ROI大小

// 插入标志
logo.copyTo(imageROI);
```

这里的`image`是目标图像，`logo`是标志图像（相对较小）。运行上述代码后，你将得到下面的图像：



1.5.3 实现原理

定义ROI的一个方法是使用cv::Rect实例。正如名称所示，它描述了一个矩形区域，方法是指明左上角的位置（构造函数的前两个参数）和矩形的尺寸（后两个参数表示宽度和高度）。这个例子中，我们利用图像和标志的尺寸来确定标志的位置，即图像的右下角。很明显，整个ROI肯定处于父图像的内部。

ROI还可以用行和列的值域来描述。值域是一个从开始索引到结束索引的连续序列（不含开始值和结束值），我们可以用cv::Range结构来表示这个概念。因此，一个ROI可以用两个值域来定义。在本例中，ROI同样可以定义如下：

```
imageROI= image(cv::Range(image.rows-logo.rows,image.rows),
                cv::Range(image.cols-logo.cols,image.cols));
```

这里，cv::Mat的operator()函数返回另一个cv::Mat实例，可在后面使用。由于图像和ROI共享了同一块图像数据，因此ROI的任何转变都会影响原始图像的相关区域。在定义ROI时数据并没有被复制，因此它的执行时间是固定的，不受ROI尺寸的影响。

要定义由图像中的一些行组成的ROI，你可用下面的代码：

```
cv::Mat imageROI= image.rowRange(start,end);
```

类似地，要定义由图像中一些列组成的ROI，你可用下面的代码：

```
cv::Mat imageROI= image.colRange(start,end);
```

1.5.4 扩展阅读

OpenCV的方法和函数包含了很多本书并不涉及的可选参数，在第一次使用某个函数时，你需要花时间看一下文档，以查清该函数支持哪些选项。一个十分常见的选项很可能被用来定义图像掩码。

使用图像掩码

OpenCV中的有些操作可以用来定义掩码。函数或方法通常对图像中所有的像素进行操作，通过定义掩码可以限制这些函数或方法的作用范围。掩码是一个8位图像，如果掩码中某个位置的值不为0，在这个位置上的操作就会起作用；如果掩码中某些像素位置的值为0，那么对图像中相应位置的操作将不起作用。例如，在调用copyTo方法时我们就可以使用掩码。这里，我们可以利用掩码只复制标志中白色的部分，如下：

```
// 在图像的右下角定义一个ROI
imageROI= image(cv::Rect(image.cols-logo.cols,
                         image.rows-logo.rows,
                       logo.cols,logo.rows));
```

```
// 把标志作为掩码（必须是灰度图像）
cv::Mat mask(logo);

// 插入标志，只复制掩码不为0的位置
logo.copyTo(imageROI,mask);
```

执行这段代码后你将得到下面这个图像：



标志的背景是黑色的（因此值为0），所以我们很容易把它作为被复制图像和掩码。当然，我们可以在程序中自己决定如何定义掩码。OpenCV中大多数基于像素的操作都可以使用掩码。

1.5.5　参阅

- 2.6节将用到row和col方法。它们是rowRange和colRange方法的特例，即开始和结束的索引是相同的，以定义一个单行或单列的ROI。

第 2 章

操作像素

本章包括以下内容：

- ❑ 访问像素值；
- ❑ 用指针扫描图像；
- ❑ 用迭代器扫描图像；
- ❑ 编写高效的图像扫描循环；
- ❑ 扫描图像并访问相邻像素；
- ❑ 实现简单的图像运算；
- ❑ 图像重映射。

2.1 简介

为了构建计算机视觉应用程序，我们需要学会访问图像内容，有时也要修改或创建图像。本章讲解如何操作图像的元素（即像素），你将学会如何扫描一个图像并处理每一个像素，还将学会如何高效地进行处理，因为即使是中等大小的图像，也能包含数十万个像素。

图像本质上就是一个由数值组成的矩阵。正因为如此，OpenCV 2使用了`cv::Mat`结构来操作图像，这在第1章已经讲过。矩阵中的每个元素表示一个像素。对于灰度图像（黑白图像），像素是8位无符号数，0表示黑色，255表示白色。对于彩色图像，需要用三原色数据来重现不同的可见色，这是因为我们人类的视觉系统是三原色的，视网膜上有三种类型的视锥细胞，它们将颜色信息传递给大脑。这意味着对于彩色图像，每个像素都要对应三个数值。在摄影和数字成像技术中，常用的主颜色通道是红色、绿色和蓝色，因此每3个8位数值组成矩阵的一个元素。

注意，8位通道通常是够用的，但有些特殊的应用程序需要用16位通道（例如医学图像）。

在第1章我们看到，OpenCV中也可以用其他类型的像素值来创建矩阵（或图像），例如整型（`CV_32U`或`CV_32S`）和浮点数（`CV_32F`）。这些类型非常有用，例如可存储图像处理过程中的中间结果。大部分操作可以使用任何类型的矩阵，还有一些操作必须使用特定的类型或特定的通道数量。因此，为了避免常见的编程错误，必须充分理解函数或方法的先决条件。

本章，我们将一直使用下面的彩色图像（另见彩图1）作为输入对象：



2.2　访问像素值

要访问矩阵中的每个独立元素，你只需要指定它的行号和列号。返回的对应元素可以是单个数值，也可以是多通道图像的数值向量。

2.2.1　准备工作

为了说明如何直接访问像素值，我们创建一个简单的函数，用它在图像中加入椒盐噪声（salt-and-pepper noise）。顾名思义，椒盐噪声是一个专门的噪声类型，它随机选择一些像素，把它们的颜色替换成白色或黑色。如果通信时出错，部分像素的值在传输时丢失，就会发生这种噪声。这里我们只是随机地选择一些像素，把它们设置为白色。

2.2.2　如何实现

创建一个接受输入图像的函数，在函数中对图像进行修改。第二个参数是需要改成白色的像素数量。

```
void salt(cv::Mat image, int n) {

  int i,j;
  for (int k=0; k<n; k++) {
```

```
        // rand()是随机数生成器
        i= std::rand()%image.cols;
        j= std::rand()%image.rows;


        if (image.type() == CV_8UC1) { // 灰度图像

          image.at<uchar>(j,i)= 255;

        } else if (image.type() == CV_8UC3) { // 彩色图像

          image.at<cv::Vec3b>(j,i)[0]= 255;
          image.at<cv::Vec3b>(j,i)[1]= 255;
          image.at<cv::Vec3b>(j,i)[2]= 255;
        }
      }
    }
```

这个函数使用一个简单的循环，执行`n`次，每次把随机选择的像素设置为255。这里用随机数生成器产生像素的列`i`和行`j`。注意，这里使用了`type`方法来区分灰度图像和彩色图像。对于灰度图像，把单个的8位数值设置为255；对于彩色图像，需要把三个主颜色通道设置为255，才能得到一个白色像素。

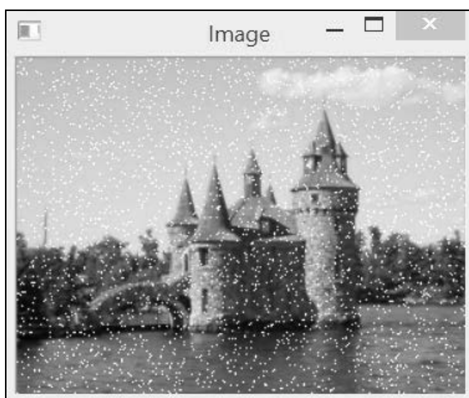
现在你可以调用这个函数，并传入已经打开的图像。参考下面的代码：

```
// 打开图像
cv::Mat image= cv::imread("boldt.jpg");

// 调用函数以添加噪声
salt(image,3000);

// 显示图像
cv::namedWindow("Image");
cv::imshow("Image",image);
```

结果图像（另见彩图2）如下：



2.2.3 实现原理

cv::Mat类包含有多种方法，可用来访问图像的各种属性：利用公共成员变量cols和rows可得到图像的列数和行数；利用cv::Mat的at(int y,int x)方法可以访问元素。在编译时必须明确方法返回值的类型。因为cv::Mat可以接受任何类型的元素，所以程序员需要指定返回值的预期类型。正因为如此，at方法被实现成一个模板方法。在调用时必须指定图像元素的类型，如：

```
image.at<uchar>(j,i)= 255;
```

有一点需要特别注意，程序员必须保证指定的类型与矩阵内的类型是一致的。at方法不会进行任何类型转换。

彩色图像的每个像素对应三个部分：红色、绿色和蓝色通道。因此包含彩色图像的cv::Mat类会返回一个向量，向量中包含三个8位的数值。OpenCV为这样的短向量定义了一种类型，即cv::Vec3b。这个向量包含三个无符号字符（unsigned character）类型的数据。因此，访问彩色像素中元素的方法如下：

```
image.at<cv::Vec3b>(j,i)[channel]= value;
```

channel索引用来指明三个颜色通道中的一个。OpenCV存储通道数据的次序是蓝色、绿色和红色（因此蓝色是通道0）。

还有类似的向量类型表示二元素向量和四元素向量（cv::Vec2b和cv::Vec4b）。此外还有针对其他元素类型的向量。例如，表示二元素浮点数类型的向量，类型名称的最后一个字母换成“f”，即cv::Vec2f。对于短整型，最后的字母换成s；对于整型，最后的字母换成i；对于双精度浮点数类型，最后的字母换成d。所有这些类型都用cv::Vec<T,N>模板类定义，其中T是类型，N是向量元素的数量。

最后一个提示，你也许会觉得奇怪，这些修改图像的函数在使用图像作为参数时，都采用了值传递的方式。之所以这样做，是因为它们在复制图像时仍共享了同一块图像数据。因此在需要修改图像内容时，图像参数没必要采用引用传递的方式。顺便说一下，编译器做代码优化时，用值传递参数的方法通常比较容易实现。

2.2.4 扩展阅读

cv::Mat类的定义采用了C++模板，因此它的通用性很强。

cv::Mat_模板类

因为每次调用都必须在模板参数中指明返回类型，所以使用cv::Mat类的at方法有时会显得冗长。如果已经知道矩阵的类型，就可以使用cv::Mat_类（cv::Mat类的模板子类）。cv::Mat_

类定义了一些新的方法，但没有定义新的数据属性，因此这两个类的指针或引用可以直接互相转换。在新方法中有一个operator()，可用它直接访问矩阵的元素。因此可以这样写代码（其中image是一个对应uchar矩阵的cv::Mat变量）：

```
// 用Mat_模板操作图像
cv::Mat_<uchar> im2(image);
im2(50,100)= 0; // 访问第50行、第100列处那个值
```


在创建cv::Mat_变量时，我们就定义了它的元素类型，因此在编译时就已经知道operator()的返回类型。使用操作符operator()和使用at方法产生的结果是完全相同的，只是前者的代码更简短。

2.2.5　参阅

- 2.3.4节解释了如何创建一个带有输入和输出参数的函数。
- 2.5节有关于该方法的效率的讨论。

2.3　用指针扫描图像

在大多数图像处理任务中，执行计算时你都需要对图像的所有像素进行扫描。需要访问的像素数量非常庞大，因此你必须采用高效的方式来执行这个任务。本节和下一节将展示几种实现高效扫描循环的方法。本节使用指针运算。

2.3.1　准备工作

为了说明图像扫描的过程，我们来做一个简单的任务：减少图像中颜色的数量。

彩色图像由三通道像素组成，每个通道表示红、绿、蓝三原色中一种颜色的亮度值，每个数值都是8位的无符号字符类型，因此颜色总数为256 × 256 × 256，即超过1600万种颜色。因此，为了降低分析的复杂性，有时需要减少图像中颜色的数量。一种实现方法是把RGB空间细分到大小相等的方块中。例如，如果把每种颜色数量减少到1/8，那么颜色总数就变为32 × 32 × 32。将旧图像中的每个颜色值划分到一个方块，该方块的中间值就是新的颜色值；新图像使用新的颜色值，颜色数就减少了。

因此基本的减色算法很简单。假设N是减色因子，将图像中每个像素的每个通道的值除以N（使用整数除法，不保留余数）。然后将结果乘以N，得到N的倍数，并且刚好不超过原始像素值。只需加上N / 2，就得到相邻的N倍数之间的中间值。对所有8位通道值重复这个过程，就会得到(256 / N) × (256 / N) × (256 / N)种可能的颜色值。

2.3.2　如何实现

减色函数的签名如下：

```
void colorReduce(cv::Mat image, int div=64);
```

用户提供一个图像和每个颜色通道的减色因子。这里的处理过程是就地进行的，也就是说，函数直接修改了输入图像的像素值。2.3.4节介绍了一个更为通用的签名，用于输入和输出参数。

处理过程很简单，只要创建一个二重循环遍历所有像素值，代码如下：

```
void colorReduce(cv::Mat image, int div=64) {

     int nl= image.rows; // 行数
     // 每行的元素数量
     int nc= image.cols * image.channels();

     for (int j=0; j<nl; j++) {

        // 取得行j的地址
        uchar* data= image.ptr<uchar>(j);

        for (int i=0; i<nc; i++) {

            // 处理每个像素 ---------------------

            data[i]=    data[i]/div*div + div/2;

            // 像素处理结束 ----------------

        } // 一行结束
    }
}
```

可以用下面的代码片段测试这个函数：

```
// 读取图像
image= cv::imread("boldt.jpg");
// 处理图像
colorReduce(image,64);
// 显示图像
cv::namedWindow("Image");
cv::imshow("Image",image);
```

执行后得到下面的图像（另见彩图3）：



2.3.3 实现原理

在彩色图像中，图像数据缓冲区的前3字节表示左上角像素的三色通道，接下来的3字节表示第1行的第2个像素，依次类推（注意OpenCV默认的通道次序为BGR）。一个宽W高H的图像所需的内存块大小为W × H × 3 uchars。然而出于性能上的考虑，我们会用几个额外的像素来填补行的长度。这是因为有些多媒体处理芯片（如Intel MMX体系）处理图片时，如果行的长度是4或8的整数倍，处理的性能就会更高。当然，这些额外的像素不显示也不保存，它们的额外数据会被忽略。OpenCV把经过填充的行的长度指定为有效宽度。如果图像没有用额外的像素填充，那么有效宽度就等于实际的图像宽度。我们已经学过，用cols和rows属性可得到图像的宽度和高度。类似地，用step数据属性可得到单位是字节的有效宽度。即使图像的类型不是uchar，step仍然能提供行的字节数。我们可以通过elemSize方法（例如一个三通道短整型的矩阵CV_16SC3，elemSize会返回6）获得像素的大小，通过nchannels方法（灰度图像为1，彩色图像为3）获得图像中通道的数量，最后，用total方法返回矩阵中的像素（即矩阵的条目）总数。

用下面的代码可获得每一行中像素值的个数：

```
int nc= image.cols * image.channels();
```

为了简化指针运算的计算过程，cv::Mat类提供了一个方法，可以直接访问图像中一个行的地址。这就是ptr方法，它是一个模板方法，返回第j行的地址：

```
uchar* data= image.ptr<uchar>(j);
```

注意在处理语句中，我们也可以采用另一种等价的做法，即利用指针运算从一列移到下一列。因此可以使用下面的代码：

```
*data= *data/div*div + div2; data++;
```

2.3.4　扩展阅读

前面介绍的减色函数只是完成任务的一种方法，此外也可以采用其他减色算法。要使函数更加通用，就要允许指定不同的输入和输出图像。如果考虑到图像数据的连续性，扫描的速度还可以提高。最后，也可以使用低层次指针运算来扫描图像缓冲区。下面分别讨论这几点。

1. 其他减色算法

前面的例子中，减色功能的实现是利用了整数除法的特性，即取不超过又最接近结果的整数，代码如下：

```
data[i]= (data[i]/div)*div + div/2;
```

减色计算也可以使用取模运算符，它得到最接近的div（每个通道的减色因子）倍数，代码如下：

```
data[i]= data[i] – data[i]%div + div/2;
```

另外还可以使用位运算符。如果把减色因子限定为2的指数，即div=pow(2,n)，那么把像素值的前面n位掩码后就得到最接近的div的倍数。可以用简单的位移操作获得掩码，代码如下：

```
// 用来截取像素值的掩码
uchar mask= 0xFF<<n; // 例如：如div=16，则mask=0xF0
```

可用下面的代码实现减色运算：

```
*data &= mask;        // 掩码
*data++ += div>>1;    // 加上div/2
```

一般来说，使用位运算的代码运行效率很高，因此在强调高效率的运行时，位运算是不二之选。

2. 使用输入和输出参数

在前面的减色函数中，直接在输入图像中进行了转换，这称为就地转换。这种做法不需要额外的图像来输出结果，可以减少内存的使用。但是有的程序不希望对原始图像进行修改，这时就必须在调用函数前备份图像。注意，对图像进行深复制的最简单方法是使用clone方法。如下面的代码：

```
// 读入图像
image= cv::imread("boldt.jpg");
// 复制图像
cv::Mat imageClone= image.clone();
// 处理图像副本
// 原始图像保持不变
colorReduce(imageClone);
// 显示结果图像
cv::namedWindow("Image Result");
cv::imshow("Image Result",imageClone);
```

如果在定义函数时允许用户选择是否要采用就地处理，就可以避免这些额外的过程。方法的签名为：

```
void colorReduce(const cv::Mat &image, // 输入图像
                 cv::Mat &result,      // 输出图像
                 int div=64);
```


注意输入图像是一个引用的const，表示这个图像不会在函数中修改。输出图像是一个引用参数，在函数中会被修改，并且返回给调用这个函数的代码。如果需要就地处理，可以在输入和输出参数中用同一个image变量：

```
colorReduce(image,image);
```

否则就可以提供一个cv::Mat实例，例如：

```
cv::Mat result;
colorReduce(image,result);
```

这里的关键是首先要检查输出图像，验证它是否分配了一定大小的数据缓冲区，以及像素类型与输入图像是否相符。cv::Mat的create方法中已经包含了这个检查过程。当你用新的大小和像素类型重新分配矩阵时，就要调用create方法。如果矩阵已有的大小和类型刚好与指定的大小和类型相同，这个方法就不会执行任何操作，也不会修改实例，而只是直接返回。

因此，函数中首先要调用create方法，构建一个大小和类型都与输入图像相同的矩阵（如果必要）：

```
result.create(image.rows,image.cols,image.type());
```

分配的内存块的大小表示为total()*elemSize()。循环过程中使用两个指针：

```
for (int j=0; j<nl; j++) {

    // 获得第j行的输入和输出的地址
    const uchar* data_in= image.ptr<uchar>(j);
    uchar* data_out= result.ptr<uchar>(j);

    for (int i=0; i<nc*nchannels; i++) {

        // 处理每个像素 ---------------------

        data_out[i]= data_in[i]/div*div + div/2;

        // 像素处理结束 ----------------

    } // 一行结束
}
```

如果输入和输出参数用了同一个图像，这个函数就与本节中前面的版本完全等效。如果输出用了另一个图像，不管在调用函数前是否已经分配了这个图像，函数都会正常运行。

3. 对连续图像的高效扫描

前面我们解释过，为了提高性能，会在图像的每行末尾用额外的像素进行填充。有趣的是，图像在去掉填充后，仍可看作一个包含W×H像素的长的一维数组。用cv::Mat的isContinuous方法可方便地判断出图像有没有被填充。如果图像中没有填充像素，它就返回true。我们还能这样测试矩阵的连续性：

```
// 检查行的长度（字节数）与“列的个数 × 单个像素”的字节数是否相等
image.step == image.cols*image.elemSize();
```

在完整的测试中，还需要检查矩阵是否只有一行。如果是，我们就说矩阵是连续的，然而总要用isContinuous方法测试连续性条件。在一些特殊的处理算法中，可以充分利用图像的连续性，在单个（更长）的循环中处理图像。处理函数就可以改为：

```
void colorReduce(cv::Mat &image, int div=64) {

     int nl= image.rows; // 行数
     int nc= image.cols * image.channels();

     if (image.isContinuous())
     {
        // 没有填充的像素
        nc= nc*nl;
        nl= 1;  // 它现在成了一个长的一维数组
     }

     // 对于连续图像，这个循环只执行一次
     for (int j=0; j<nl; j++) {

          uchar* data= image.ptr<uchar>(j);

          for (int i=0; i<nc; i++) {

            // 处理每个像素 ---------------------

            data[i]= data[i]/div*div + div/2;

            // 处理像素完毕 ----------------

          } // 一行结束
     }
}
```

如果连续性测试结果表明图像中没有填充像素，我们就把宽度设为1，高度设为$W \times H$，从而去除外层的循环。注意，这里还需要用reshape方法。本例中需要这样写：

```
if (image.isContinuous())
{
   // 没有填充像素
   image.reshape(1,   // 新的通道数
```

```
                  1); // 新的行数
    }

    int nl= image.rows; // 行数
    int nc= image.cols * image.channels();
```


用reshape方法修改矩阵的维数，你不需要复制内存或重新分配内存。第一个参数是新的通道数，第二个参数是新的行数。列数会进行相应的修改。

在这些实现方式中，内层循环按顺序处理图像中的所有像素。在同一个循环中同时扫描多个小图像时，采用这种方法很有好处。

4. 低层次指针算法

在cv::Mat类中，图像数据是存放在无符号字符型的内存块中的。其中data属性表示内存块的第一个元素的地址，它会返回一个无符号字符型的指针。要从图像的起点开始循环，你可以用如下代码：

```
uchar *data= image.data;
```

利用有效宽度来移动行指针，可以从一行移到下一行。代码如下：

```
data+= image.step; // 下一行
```

用step方法可得到一行的总字节数（包括填充像素）。通常可以用下面的方法，得到第j行、第i列的像素的地址：

```
// (j,i)像素的地址，即&image.at(j,i)
data= image.data+j*image.step+i*image.elemSize();
```

然而，尽管这种处理方法在上述例子中能起作用，但是我们并不推荐采用。

2.3.5 参阅

- 2.5节讨论各种扫描方法的效率。

2.4 用迭代器扫描图像

在面向对象编程时，我们通常用迭代器对数据集合进行循环遍历。迭代器是一种类，专门用于遍历集合的每个元素，隐藏了遍历过程的具体细节。信息隐藏原则的应用，使扫描集合的过程变得更加容易和安全。并且不管使用哪种类型的集合，它都能提供类似的形式。标准模板库（STL）对每个集合类都定义了对应的迭代器类，OpenCV也提供了cv::Mat的迭代器类，并且与C++ STL中的标准迭代器兼容。

2.4.1　准备工作

本节中我们仍使用2.3节的减色程序作为例子。

2.4.2　如何实现

要得到cv::Mat实例的迭代器，首先要创建一个cv::MatIterator_对象。跟cv::Mat_类似，这个下划线表示它是一个模板子类。因为图像迭代器是用来访问图像元素的，所以必须在编译时就明确返回值的类型。可以这样定义迭代器：

```
cv::MatIterator_<cv::Vec3b> it;
```

此外还可以使用在Mat_模板类内部定义的iterator类型：

```
cv::Mat_<cv::Vec3b>::iterator it;
```

然后就可以使用常规的迭代器方法begin和end对像素进行循环遍历了。不同之处在于它们仍然是模板方法。现在，减色函数可以这样编写：

```
void colorReduce(cv::Mat &image, int div=64) {

     // 在初始位置获得迭代器
     cv::Mat_<cv::Vec3b>::iterator it=
               image.begin<cv::Vec3b>();
     // 获得结束位置
     cv::Mat_<cv::Vec3b>::iterator itend=
               image.end<cv::Vec3b>();

     // 循环遍历所有像素
     for ( ; it!= itend; ++it) {

         // 处理每个像素 ---------------------

        (*it)[0]= (*it)[0]/div*div + div/2;
        (*it)[1]= (*it)[1]/div*div + div/2;
        (*it)[2]= (*it)[2]/div*div + div/2;

         // 像素处理结束 ----------------
    }
}
```

注意这里处理的是一个彩色图像，因此迭代器返回cv::Vec3b实例。你可以用取值运算符[]访问每个颜色通道的元素。

2.4.3　实现原理

不管扫描的是哪种类型的集合，使用迭代器时总是遵循同样的模式。

首先你要使用合适的专用类创建迭代器对象，在本例中是cv::Mat_<cv::Vec3b>::iterator（或cv::MatIterator_<cv::Vec3b>）。

然后可以用begin方法，在开始位置（本例中为图像的左上角）初始化迭代器。对于cv::Mat实例，可以使用image.begin<cv::Vec3b>()。还可以在迭代器上使用数学计算，例如若要从图像的第二行开始，可以用image.begin<cv::Vec3b>()+image.cols初始化cv::Mat迭代器。获得集合结束位置的方法也类似，只是改用end方法。但是，用end方法得到的迭代器已经超出了集合范围，因此必须在结束位置停止迭代过程。结束的迭代器也能使用数学计算，例如，如果你想在最后一行前就结束迭代，可使用image.end<cv::Vec3b>()-image.cols。

初始化迭代器后，建立一个循环遍历所有元素，直到与结束迭代器相等。典型的while循环就像这样：

```
while (it!= itend) {

   // 处理每个像素 ---------------------


   // 像素处理结束 ---------------------

   ++it;
}
```

你可以用运算符++来移动到下一个元素，也可以指定更大的步幅。例如用it+=10，对每10个像素处理一次。

最后，在循环内部使用取值运算符*来访问当前元素，你可以用它来读（例如element= *it;）或写（例如*it= element;）。你也可以创建常量迭代器，用作对常量cv::Mat的引用，或者表示当前循环不修改cv::Mat实例。常量迭代器的定义如下：

```
cv::MatConstIterator_<cv::Vec3b> it;
```

或者：

```
cv::Mat_<cv::Vec3b>::const_iterator it;
```

2.4.4 扩展阅读

本节中，用begin和end模板方法可获得迭代器的开始和结束位置。2.2节讲过，我们还可以用对cv::Mat_实例的引用来获得迭代器的开始和结束位置，这样就不需要在begin和end方法中指定迭代器的类型了，因为在创建cv::Mat_引用时迭代器类型已被指定。

```
cv::Mat_<cv::Vec3b> cimage(image);
cv::Mat_<cv::Vec3b>::iterator it= cimage.begin();
cv::Mat_<cv::Vec3b>::iterator itend= cimage.end();
```

2.4.5　参阅

- 2.5节讨论迭代器在扫描图像时的效率。
- 如果你不熟悉面向对象编程中迭代器的概念，不知道在ANSI C++中如何实现迭代器，可阅读STL迭代器的教程。在网上搜索关键字“STL迭代器”，你会发现很多此类内容。

2.5　编写高效的图像扫描循环

本章前面几节介绍了为处理像素而扫描图像的几种方法。本节我们来比较一下这些方法的效率。

在编写图像处理函数时，你需要充分考虑运行效率。在设计函数时，你要经常检查代码的运行效率，找出处理过程中可能使程序变慢的瓶颈。

但是有一点非常重要，除非确实必要，不要以牺牲程序的清晰度来优化性能。简洁的代码总是更容易调试和维护。只有对程序效率至关重要的代码段，才需要进行重度优化。

2.5.1　如何实现

为了衡量函数或代码段的运行时间，OpenCV有一个非常实用的函数，即`cv::getTickCount()`，该函数返回从最近一次电脑开机到当前的时钟周期数。因为我们希望得到以秒为单位的代码运行时间，所以要使用另一个方法，即`cv::getTickFrequency()`，这个方法返回每秒的时钟周期数。为了获得某个函数（或代码段）的运行时间，通常需使用这样的程序模板：

```
const int64 start = cv::getTickCount();
colorReduce(image); // 调用函数
// 经过的时间（单位：秒）
double duration = (cv::getTickCount()-start)/
                              cv::getTickFrequency();
```

2.5.2　实现原理

本章的`colorReduce`函数有几种不同的实现方式，这里我们列出每种方式的运行时间，实际的数据跟使用的电脑有关（我们使用配置64位Intel Core i7、主频2.40GHz的电脑）。观察运行时间的相对差距更有意义。此外，测试结果也跟生成可执行文件的具体编译器有关。我们采用4288像素 × 2848像素的图像，测试减色操作的平均运行时间。

首先，我们来比较2.3.4节描述的三种减色运算方法。有趣的是，使用了位运算符的方法用时9.5 ms，要比其他方法快得多；使用整数除法的方法用时26 ms；使用取模运算符的方法用时33 ms。

由此可见，最快和最慢的速度相差竟超过3倍！因此，要在图像循环中计算出结果，花些时间找出效率最高的方法十分重要，其净影响会非常明显。

如果需要重新分配输出图像而不是就地处理，运行时间就变为29 ms。增加的时间代表内存分配的开销。

对于可以预先计算的数值，要避免在循环中做重复计算，这样很明显会浪费时间。例如在减色函数中使用下面的内层循环：

```
int nc= image.cols * image.channels();
uchar div2= div>>1;

for (int i=0; i<nc; i++) {
```

然后改成这样：

```
for (int i=0; i<image.cols * image.channels(); i++) {
// . . .
*data++ += div>>1;
```

在前面的代码循环中，你需要反复计算一行的元素个数和`div>>1`的结果，其运行时间是52 ms，明显比修改前的版本（26 ms）要慢。但是要注意，有些编译器能够对此类循环进行优化，仍会生成高效的代码。

2.4节讨论了使用迭代器的减色函数，它的运行时间更长，为52 ms。使用迭代器的主要目的是简化图像扫描过程，降低出错的可能性。

为了进行完整的测试，我们实现了用`at`方法访问像素的函数。这种实现方式的主循环如下：

```
for (int j=0; j<nl; j++) {
  for (int i=0; i<nc; i++) {

    // 处理每个像素 ---------------------

    image.at<cv::Vec3b>(j,i)[0]=
               image.at<cv::Vec3b>(j,i)[0]/div*div + div/2;
    image.at<cv::Vec3b>(j,i)[1]=
               image.at<cv::Vec3b>(j,i)[1]/div*div + div/2;
    image.at<cv::Vec3b>(j,i)[2]=
               image.at<cv::Vec3b>(j,i)[2]/div*div + div/2;

    // 像素处理结束 ----------------

  } // 一行结束
}
```

这种实现方法用时53 ms，要慢很多。该方法应该在需要随机访问像素的时候使用，绝不要在扫描图像时使用。

即使处理的元素总数相同，使用较短的循环和多条语句通常也要比使用较长的循环和单条语句的运行效率高。类似地，如果你要对一个像素执行N个不同的计算过程，那就在单个循环中执行全部计算，而不是写N个连续的循环，每个循环执行一个计算。

我们还做过连续性测试，针对连续图像生成一个循环，而不是对行和列运行常规的二重循环。例如，对于我们测试中用到的非常大的图像，这种优化效果不明显（从26 ms变为25 ms）。但通常情况下，采用这种策略是一个非常好的做法，因为它会明显提升速度。

2.5.3　扩展阅读

还有一个提高程序运行效率的方法是采用多线程，尤其是在使用多核处理器时。OpenMP和Intel线程构建模块（Threading Building Block，TBB）是两个流行的并发编程API，用于创建和管理线程。而且现在C++11本身就支持多线程。

2.5.4　参阅

- 2.7.4节介绍了一种减色函数的实现方法，它使用了OpenCV 2算法图像运算符，运行时间为25 ms。
- 4.2节介绍了一种基于速查表的减色函数实现方法，它的理念是预先计算所有减少亮度的值，运行时间为22 ms。

2.6　扫描图像并访问相邻像素

在图像处理中计算像素值时，经常需要用它的相邻像素的值。如果相邻像素在上一行或下一行，就需要同时扫描图像的多行。本节介绍实现方法。

2.6.1　准备工作

为了便于说明问题，我们使用一个锐化图像的处理函数。它基于拉普拉斯算子（将在第6章中讨论）。在图像处理领域有一个众所周知的结论：如果从图像中减去拉普拉斯算子部分，图像的边缘就会放大，因而图像会变得更加尖锐。

用以下方法计算锐化的数值：

```
sharpened_pixel= 5*current-left-right-up-down;
```

这里的left是与当前像素相邻的左侧像素，up是在上一行的相邻像素，以此类推。

2.6.2 如何实现

这里不能使用就地处理，使用者必须提供一个输出图像。图像扫描中使用了三个指针，一个表示当前行，一个表示上面的行，另外一个表示下面的行。另外，在计算每一个像素时都需要访问与它相邻的像素，因此有些像素的值是无法计算的，包括第一行、最后一行、第一列、最后一列的像素。这个循环可以这样写：

```
void sharpen(const cv::Mat &image, cv::Mat &result) {

   // 判断是否需要分配图像数据。如果需要，就分配
  result.create(image.size(), image.type());
  int nchannels= image.channels(); // 获得通道数

   // 处理所有行（除了第一行和最后一行）
  for (int j= 1; j<image.rows-1; j++) {

    const uchar* previous=
        image.ptr<const uchar>(j-1);     // 上一行
    const uchar* current=
        image.ptr<const uchar>(j);       // 当前行
    const uchar* next=
        image.ptr<const uchar>(j+1);     // 下一行

    uchar* output= result.ptr<uchar>(j); // 输出行

    for (int i=nchannels; i<(image.cols-1)*nchannels; i++) {

       *output++= cv::saturate_cast<uchar>(
                  5*current[i]-current[i-nchannels]-
                  current[i+nchannels]-previous[i]-next[i]);
    }
  }

  // 把未处理的像素设为0
  result.row(0).setTo(cv::Scalar(0));
  result.row(result.rows-1).setTo(cv::Scalar(0));
  result.col(0).setTo(cv::Scalar(0));
  result.col(result.cols-1).setTo(cv::Scalar(0));
}
```

注意这个函数是如何同时适应灰度图像和彩色图像的。如果我们在测试用的灰度图像上执行该函数，将得到如下结果：



2.6.3　实现原理

为了访问在上一行和下一行的相邻像素，只需定义额外的指针，并与当前行的指针一起递增，然后就可以在扫描循环内访问上下行的指针了。

在计算输出像素的值时，我们调用了`cv::saturate_cast`模板函数，并传入运算结果。这是因为计算像素的数学表达式的结果经常超出允许的范围（即小于0或大于255）。使用这个函数可把结果调整到8位无符号数的范围内，具体做法是把小于0的数值调整为0，大于255的数值调整为255。这就是`cv::saturate_cast<uchar>`函数的作用。如果输入参数是浮点数，就会得到最接近的整数。可以在调用这个函数时显式地指定其他数据类型，以确保结果在该数据类型定义的范围之内。

由于边框上的像素没有完整的相邻像素，因此不能用前面的方法计算，需要另行处理。这里我们简单地把它们设置为0。有时也可以对这些像素做特殊的计算，但是大多数情况下，花时间处理这些极少数像素是没有意义的。在本例中，我们在把边框的像素设置为0时用到了两个特殊的方法，即`row`和`col`。这两个方法返回一个特殊的`cv::Mat`实例，它包含一个单行ROI（或单列ROI），具体范围取决于参数（第1章讨论过兴趣区域）。这里没有进行复制，因为只要这个一维矩阵的元素被修改，原始图像也会修改。我们用`setTo`方法来实现这个功能，此方法对矩阵中所有元素赋值。看下面的语句：

```
result.row(0).setTo(cv::Scalar(0));
```

这个语句把结果图像第一行的所有像素设置为0。对于三通道彩色图像，需要使用`cv::Scalar(a,b,c)`来指定三个数值，分别对像素的每个通道赋值。

2.6.4 扩展阅读

在对像素邻域进行计算时，通常用一个核心矩阵来表示。这个核心矩阵展现了为得到预期结果，如何将计算相关的像素组合起来。针对本节使用的锐化滤波器，核心矩阵可以是这样的：

0	–1	0
–1	5	–1
0	–1	0

除非另有说明，当前像素用核心矩阵中心单元格表示。核心矩阵中每个单元格表示相关像素的乘法系数，像素应用核心矩阵得到的结果，就是这些乘积的累加。核心矩阵的大小就是邻域的大小（这里是3 × 3）。从这个描述可以看出，根据锐化滤波器的要求，水平和垂直方向的四个相邻像素被乘以–1，当前像素被乘以5。在图像上应用核心矩阵，不只是为了描述方便，它是信号处理中卷积概念的基础。核心矩阵定义了一个用于图像的滤波器。

鉴于滤波是图像处理中常见的操作，OpenCV专门为此定义了一个函数，即`cv::filter2D`。要使用这个函数，只需要定义一个内核（以矩阵的形式），调用函数并传入图像和内核，即可返回滤波后的图像。因此，使用这个函数可以很容易地重新定义锐化函数：

```
void sharpen2D(const cv::Mat &image, cv::Mat &result) {

   // 构造内核（所有入口都初始化为0）
   cv::Mat kernel(3,3,CV_32F,cv::Scalar(0));
   // 对内核赋值
   kernel.at<float>(1,1)= 5.0;
   kernel.at<float>(0,1)= -1.0;
   kernel.at<float>(2,1)= -1.0;
   kernel.at<float>(1,0)= -1.0;
   kernel.at<float>(1,2)= -1.0;

   // 对图像滤波
   cv::filter2D(image,result,image.depth(),kernel);
}
```

这种实现方式得到的结果，与前面的完全相同（执行效率也相同）。如果处理彩色图像，三个通道可以应用同一个内核。注意，使用大内核的`filter2D`函数是特别有利的，因为这种情况下它使用了更高效的算法。

2.6.5 参阅

- 第6章更详细地解释了图像滤波的概念。

2.7　实现简单的图像运算

图像就是普通的矩阵，可以进行加、减、乘、除运算，因此可以用多种不同的方式组合图像。OpenCV提供了很多图像算法运算符，本节将讨论它们的用法。

2.7.1　准备工作

我们使用算法运算符，将第二个图像与输入图像进行组合。下面就是第二个图像：

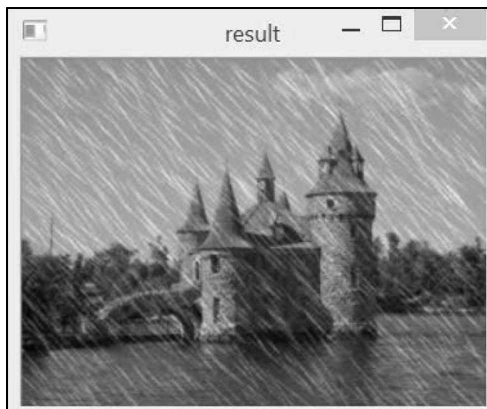


2.7.2　如何实现

这里我们把两个图像相加，用于创建特效图或覆盖图像中的信息。我们可以使用`cv::add`函数来实现相加功能。现在我们想得到加权和，因此使用更精确的`cv::addWeighted`函数：

```
cv::addWeighted(image1,0.7,image2,0.9,0.,result);
```

操作的结果是一个新图像，如下图所示：



2.7.3 实现原理

所有二进制运算函数的用法都一样：提供两个输入参数，指定一个输出参数。有些情况下还可以指定加权系数，作为运算时的缩放因子。每个函数都可以有多种格式，cv::add是典型的具有多种格式的函数：

```
// c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC);
// c[i]= a[i]+k;
cv::add(imageA,cv::Scalar(k),resultC);
// c[i]= k1*a[1]+k2*b[i]+k3;
cv::addWeighted(imageA,k1,imageB,k2,k3,resultC);
// c[i]= k*a[1]+b[i];
cv::scaleAdd(imageA,k,imageB,resultC);
```

有些函数还可以指定一个掩码：

```
// if (mask[i]) c[i]= a[i]+b[i];
cv::add(imageA,imageB,resultC,mask);
```

使用了掩码后，操作就只会在掩码值非空的像素上执行（掩码必须是单通道的）。看一下cv::subtract、cv::absdiff、cv::multiply和cv::divide等函数的多种格式。此外还有位运算符（对像素的二进制数值进行按位运算）：cv::bitwise_and、cv::bitwise_or、cv::bitwise_xor和cv::bitwise_not。cv::min和cv::max运算符也非常实用，它们可找到每个元素中最大或最小的像素值。

在所有场合都要使用cv::saturate_cast函数（见2.6节），以确保结果在预定的像素值范围之内（避免上溢或下溢）。

这些图像必定有相同的大小和类型（如果与输入图像大小不匹配，输出图像会重新分配）。由于运算是逐个元素进行的，因此我们可以把其中的一个输入图像用作输出图像。

还有一些运算符使用单个输入图像：cv::sqrt、cv::pow、cv::abs、cv::cuberoot、cv::exp和cv::log。事实上，无论需要对图像像素做任何运算，OpenCV几乎都有相应的函数。

2.7.4 扩展阅读

对于cv::Mat实例或者实例中的个别通道，也可以使用普通的C++运算符。下面两节解释如何实现。

1. 重载图像运算符

OpenCV 2中，大多数运算函数都有对应的重载运算符。因此调用cv::addWeighted的语句可以写成：

```
result= 0.7*image1+0.9*image2;
```

这种代码更加紧凑也更容易阅读。这两种计算加权和的方法是等效的。特别指出，这两种方法都会调用`cv::saturate_cast`函数。

大部分C++运算符都已被重载。其中包括：位运算符`&`、`|`、`^`和`~`；函数`min`、`max`和`abs`；比较运算符`<`、`<=`、`==`、`!=`、`>`和`>=`，它们返回一个8位的二值图像。此外还有矩阵乘法`m1*m2`（其中m1和m2都是`cv::Mat`实例）、矩阵求逆`m1.inv()`、变位`m1.t()`、行列式`m1.determinant()`、求范数`v1.norm()`、叉乘`v1.cross(v2)`、点乘`v1.dot(v2)`，等等。在理解这点后，你就会使用相应的组合赋值符了（例如`+=`运算符）。

在2.4节我们讨论了一个减色函数，它使用循环来扫描图像的像素并对像素进行运算操作。利用本节的知识，我们可以使用针对输入图像的运算符简单地重写这个函数：

```
image=(image&cv::Scalar(mask,mask,mask))
             +cv::Scalar(div/2,div/2,div/2);
```

由于我们操作的是彩色图像，因此使用了`cv::Scalar`。用2.4节中的方法做测试，得到结果是53毫秒。使用图像运算符可以简化代码，提高开发效率，因此应该考虑在大多数场合采用。

2. 分割图像通道

有时我们需要分别处理图像中的不同通道，例如只对图像中的一个通道执行某个操作。这当然可以通过图像扫描循环实现，但也可以使用`cv::split`函数，将图像的三个通道分别复制到三个不同的`cv::Mat`实例中。假设我们要把一个雨景图只加到蓝色通道中，可以这样实现：

```
// 创建三个图像的向量
std::vector<cv::Mat> planes;
// 分割一个三通道图像为三个单通道图像
cv::split(image1,planes);
// 加到蓝色通道上
planes[0]+= image2;
// 合并三个单通道图像为一个三通道图像
cv::merge(planes,result);
```

这里的`cv::merge`函数执行反向的操作，即用三个单通道图像创建一个彩色图像。

2.8　图像重映射

在本章的前面几节，我们学习了如何读取和修改图像的像素值，最后一节我们来学习如何通过移动像素修改图像的外观。这个过程不会修改图像值，而是把每个像素的位置重新映射到新的位置。这可用来创建图像特效，或者修正因镜片等原因导致的图像扭曲。

2.8.1 如何实现

要使用OpenCV的remap函数，首先需要定义在重映射处理中使用的映射参数，然后把映射参数应用到输入图像。很明显，定义映射参数的方式将决定产生的效果。这里我们定义一个转换函数，在图像上创建波浪形效果：


```
// 重映射图像，创建波浪形效果
void wave(const cv::Mat &image, cv::Mat &result) {

  // 映射参数
  cv::Mat srcX(image.rows,image.cols,CV_32F);
  cv::Mat srcY(image.rows,image.cols,CV_32F);

  // 创建映射参数
  for (int i=0; i<image.rows; i++) {
    for (int j=0; j<image.cols; j++) {

      // (i,j)像素的新位置
      srcX.at<float>(i,j)= j; // 保持在同一列
                // 原来在第i行的像素，现在根据一个正弦曲线移动
      srcY.at<float>(i,j)= i+5*sin(j/10.0);
    }
  }

  // 应用映射参数
  cv::remap(image, result, srcX, srcY, cv::INTER_LINEAR);
}
```

得到结果如下：



2.8.2 实现原理

重映射是为了修改像素的位置，生成一个新版本的图像。为了构建新图像，需要知道目标图

像的每个像素在源图像中的原始位置。因此我们需要这样的映射函数，它能根据像素的新位置得到像素的原始位置。这个转换过程描述了如何把新图像的像素映射回原始图像，因此称为反向映射。在OpenCV中，可以用两个映射参数来说明反向映射：一个针对x坐标，另一个针对y坐标。它们都用浮点数型的`cv::Mat`实例来表示：

```
// 映射参数
cv::Mat srcX(image.rows,image.cols,CV_32F); // x方向
cv::Mat srcY(image.rows,image.cols,CV_32F); // y方向
```

这些矩阵的大小决定了目标图像的大小。目标图像中`(i,j)`像素的值，可以用下面的代码从原始图像获得：

```
( srcX.at<float>(i,j) , srcY.at<float>(i,j) )
```

例如我们在第1章中展示的简单的图像翻转效果，可以用下面的映射参数创建：

```
// 创建映射参数
for (int i=0; i<image.rows; i++) {
  for (int j=0; j<image.cols; j++) {

    // 水平翻转
    srcX.at<float>(i,j)= image.cols-j-1;
    srcY.at<float>(i,j)= i;
  }
}
```

只要简单地调用OpenCV的`remap`函数，即可生成结果图像：

```
// 应用映射参数
cv::remap(image,             // 源图像
          result,            // 目标图像
          srcX,              // x方向映射
          srcY,              // y方向映射
          cv::INTER_LINEAR); // 插值法
```

有趣的是，这两个映射参数包含的值是浮点数。因此目标图像中的像素可以映射回一个非整数的值（即处在两个像素之间的位置），这使我们可以随意地定义映射参数，非常实用。例如在前面的重映射例子中，我们用了一个`sinusoidal`函数进行转换。但是，这也导致必须在真实的像素之间插入虚拟像素的值。可以采用不同的方法实现像素插值，并且可用`remap`函数的最后一个参数，来表示选择了哪种方法。像素插值是图像处理中一个重要的概念，将在第6章中讨论。

2.8.3　参阅

- 6.2.3节解释了像素插值的概念。
- 10.2节使用重映射来校正图像中的镜片扭曲。
- 10.5节使用透视图像变形来构建图像全景。

第 3 章

用类处理彩色图像

3

本章包括以下内容：

- 在算法设计中使用策略模式；
- 用控制器设计模式实现功能模块间通信；
- 转换颜色表示法；
- 用色调、饱和度、亮度表示颜色。

3.1 简介

优秀的计算机视觉程序来源于良好的编程实践。开发无bug的应用程序只是最基本的要求，真正好的程序能让你和你在团队在面临新的需求时很容易地对程序进行修改和升级。本章介绍如何充分利用面向对象编程的理念开发高质量的软件程序，并将特别介绍几种重要的设计模式，以便开发易于测试、维护和重用的应用组件。

在软件工程中，设计模式是一个广为人知的概念。总体来说，一个设计模式就是一个可靠、可重用的解决方案，用来解决软件设计中常见的一般性问题。现在已经有很多软件设计模式，并且都有很详细的文档资料。优秀的开发人员应该在实践中掌握现有的设计模式。

本章还有一个目的，就是介绍如何处理图像的颜色。本章贯穿使用的例子将展示如何检测某种特定颜色的像素，最后两节解释如何使用不同的色彩空间。

3.2 在算法设计中使用策略模式

策略设计模式的目的就是把算法封装进类。封装后，算法之间互相替换，或者把几个算法组合起来进行更复杂的处理，都会更加容易。而且这种模式能够尽可能地将算法的复杂性隐藏在一个直观的编程接口之后，因而有利于算法的部署。

3.2.1　准备工作

假设我们要构建一个简单的算法，用来识别图像中具有某种颜色的所有像素。这个算法必须输入一个图像和一个颜色，并且返回一个二值图像，显示具有指定颜色的像素。在运行算法前还要指定一个参数，即我们能接受的颜色的公差。

3.2.2　如何实现

一旦用策略设计模式把算法封装进类，就可以通过创建类的实例来部署算法，实例通常是在程序初始化的时候创建的。在运行构造函数时，类的实例会用默认值初始化算法的各种参数，以便它能立即进入可用状态。我们还可以用适当的方法来读写算法的参数值。在GUI程序中，可以用多种部件（文本框、滑动条等）显示和修改参数，用户操作起来很容易。

下一节将展示一个策略类的结构。这里先看一个部署和使用它的例子。我们写一个简单的主函数，调用颜色检测算法：

```
int main()
{
   // 1. 创建图像处理器对象
   ColorDetector cdetect;

   // 2. 读取输入的图像
   cv::Mat image= cv::imread("boldt.jpg");
   if (image.empty())
      return 0;

   // 3. 设置输入参数
   cdetect.setTargetColor(230,190,130); // 这里表示蓝天

   cv::namedWindow("result");

   // 4. 处理图像并显示结果
   cv::imshow("result",cdetect.process(image));

   cv::waitKey();

   return 0;
}
```

运行这个程序，检测第2章用过的彩色城堡图中的蓝天，输出结果如下页图所示。

这里白色像素表示检测到指定的颜色，黑色表示没有检测到。

很明显，封装进这个类的算法相对简单（下面我们会看到，它只是组合了一个扫描循环和一个公差参数）。当算法的实现过程更加复杂，步骤繁多，并且包含多个参数时，策略设计模式会真正显示出强大的功能。



3.2.3　实现原理

这个算法的核心过程非常简单，它只是对每个像素进行循环扫描，把它的颜色和目标颜色做比较。利用在2.3节学过的知识，可以这样写这个循环：

```
// 取得迭代器
cv::Mat_<cv::Vec3b>::const_iterator it=
                        image.begin<cv::Vec3b>();
cv::Mat_<cv::Vec3b>::const_iterator itend=
                        image.end<cv::Vec3b>();
cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

// 对于每个像素
for ( ; it!= itend; ++it, ++itout) {

       // 比较与目标颜色的差距
       if (getDistanceToTargetColor(*it)<=maxDist) {
            *itout= 255;
       } else {
            *itout= 0;
       }
}
```

cv::Mat 类型的变量image表示输入图像，result表示输出的二值图像。因此首先要创建迭代器，这样扫描循环就很容易实现了。在每个迭代步骤中计算当前像素的颜色与目标颜色的差距，检查它是否在公差（maxDist）范围之内。如果是，就在输出图像中赋值255（白色）；否则就赋值0（黑色）。这里用getDistanceToTargetColor方法来计算与目标颜色的差距。此外还有其他方法可计算这个差距，例如计算包含RGB颜色值的三个向量之间的欧几里得距离。为了简化计算过程，这里我们只是把RGB值差距的绝对值（也称为城区距离）进行累加。注意，在现代体系结构中浮点数的欧几里德距离的计算速度可能比简单的城区距离更快，在做设计时我们也需要考虑这点。另外，为了增加灵活性，我们依据getColorDistance方法来编写getDistance

ToTargetColor方法：

```
// 计算与目标颜色的差距
int getDistanceToTargetColor(const cv::Vec3b& color) const {
  return getColorDistance(color, target);
}

// 计算两个颜色之间的城区距离
int getColorDistance(const cv::Vec3b& color1,
                     const cv::Vec3b& color2) const {
  return abs(color1[0]-color2[0])+
              abs(color1[1]-color2[1])+
              abs(color1[2]-color2[2]);
}
```

我们用cv::Vec3d存储三个无符号字符型，即颜色的RGB值。变量target表示指定的目标颜色，是算法类的成员变量。现在我们来定义处理方法。用户提供一个输入图像，图像扫描完成后即返回结果：

```
cv::Mat ColorDetector::process(const cv::Mat &image) {

    // 必要时重新分配二值映像
    // 与输入图像的尺寸相同，不过是单通道
    result.create(image.size(),CV_8U);
    // 在这里放前面的处理循环
     ...

    return result;
}
```

在调用这个方法时，一定要检查输出图像（包含二值映像）是否需要重新分配，以匹配输入图像的尺寸。因此我们使用了cv::Mat的create方法。注意，只有在指定的尺寸或深度与当前图像结构不匹配时，它才会进行重新分配。

我们已经定义了核心的处理方法，下面就看一下为了部署该算法，还需要添加哪些额外方法。前面我们已经明确了算法需要的输入和输出数据，因此，首先要定义类的属性来存储这些数据：

```
class ColorDetector {

  private:

    // 允许的最小差距
    int maxDist;

    // 目标颜色
    cv::Vec3b target;

    // 存储二值映像结果的图像
    cv::Mat result;
```

要为封装了算法的类（已命名为ColorDetector）创建实例，就需要定义一个构造函数。要知道使用策略设计模式的原因之一，就是让算法的部署尽可能简单，最简单的构造函数当然是空函数。它会创建一个算法类的实例，并处于有效状态。然后我们在构造函数中初始化全部输入参数，设置为默认值（或采用通常会带来好结果的值）。这里我们认为通常能接受的公差参数是100。同时设置默认的目标颜色，这里选用黑色（选用黑色没有什么特别的原因），总的原则是要确保输入值可预期并且有效。

```
// 空构造函数
// 在此初始化默认参数
ColorDetector() : maxDist(100), target(0,0,0) {}
```


此时，创建该算法类的用户可以立即调用处理方法并传入一个有效的图像，然后得到一个有效的输出。这是策略模式的另一个目的，即保证只要参数正确，算法就能正常运行。用户显然希望使用个性化设置，可以用相应的设计方法和获取方法来实现这个功能。首先要实现color公差参数的定制：

```
// 设置颜色差距的阈值
// 阈值必须是正数，否则就置为0
void setColorDistanceThreshold(int distance) {

   if (distance<0)
      distance=0;
   maxDist= distance;
}

// 取得颜色差距的阈值
int getColorDistanceThreshold() const {

   return maxDist;
}
```

注意，我们首先检查了输入的合法性。再次强调，这是为了确保算法运行的有效性。目标颜色可以用类似的方法进行设置：

```
// 设置需要检测的颜色
void setTargetColor(uchar blue,
                    uchar green,
                    uchar red) {

  // 次序为BGR
  target = cv::Vec3b(blue, green, red);
}

  // 设置需要检测的颜色
void setTargetColor(cv::Vec3b color) {

target= color;
}
```

```
  // 取得需要检测的颜色
cv::Vec3b getTargetColor() const {

  return target;
}
```

这次我们提供了setTargetColor方法的两种定义。第一个版本用三个参数表示三个颜色组件，第二个版本用cv::Vec3b保存颜色值。再次强调，这么做的目的就是让算法类更便于使用，用户只需要选择最合适的设值函数。

3.2.4　扩展阅读

本节介绍了使用策略设计模式把算法封装进类中的理念。例子中的算法可识别出图像中与指定目标颜色足够接近的像素。在过程中已经完成了计算步骤。另外我们可用函数对象来补充策略设计模式。

1. 计算两个颜色向量间的距离

要计算两个颜色向量间的距离，可使用这个简单的公式：

```
return abs(color[0]-target[0])+
       abs(color[1]-target[1])+
       abs(color[2]-target[2]);
```

然而，OpenCV中也有计算向量的欧几里德范数的函数，因此我们也可以这样计算距离：

```
return static_cast<int>(
   cv::norm<int,3>(cv::Vec3i(color[0]-target[0],
                             color[1]-target[1],
                             color[2]-target[2])));
```

改用这种方式定义getDistance方法后，得到的结果与原来非常接近。这里我们之所以使用cv::Vec3i（三个向量的整型数组），是因为减法运算得到的是整数值。

还有一点非常有趣，回顾一下第2章的内容，我们会发现OpenCV中矩阵和向量等数据结构定义了基本的算术运算符。因此有人会想这样计算距离：

```
return static_cast<int>(
   cv::norm<uchar,3>(color-target)); // 错误!
```

这种做法看上去好像是对的，但实际上它是错误的。这是因为，为了确保结果在输入数据类型的范围之内（这里是uchar），这些运算符通常都调用了saturate_cast（见2.6节）。因此在target的值比color大的时候，结果就会是0 而不是负数。正确的做法应该是：

```
cv::Vec3b dist;
cv::absdiff(color,target,dist);
return cv::sum(dist)[0];
```

然而，在计算两个数组间的距离时调用了两个函数，因此效率并不高。

2. 使用OpenCV函数

本节中我们采用在循环中使用迭代器的方法来进行计算。还有一种做法是调用OpenCV的系列函数，也能得到一样的结果。因此这个检测颜色的方法可以这样写：

```
cv::Mat ColorDetector::process(const cv::Mat &image) {

        cv::Mat output;
        // 计算与目标颜色的距离的绝对值
        cv::absdiff(image,cv::Scalar(target),output);
        // 把通道分割进3个图像
        std::vector<cv::Mat> images;
        cv::split(output,images);
        // 3个通道相加（这里可能出现饱和的情况）
        output= images[0]+images[1]+images[2];
        // 应用阈值
        cv::threshold(output,                  // 输入图像
                      output,                  // 输出图像
                      maxDist,                 // 阈值（必须 < 256）
                      255,                     // 最大值
        cv::THRESH_BINARY_INV); // 阈值化模式

        return output;
}
```

该方法使用了absdiff函数计算图像的像素与标量值之间差距的绝对值。该函数的第二个参数也可以不用标量值，改用另一个图像，这样就可以逐个像素地计算差距。因此两个图像的尺寸必须相同。然后我们用split函数提取出存放差距的图像的单个通道（参见2.7.4节），以便求和。注意，累加值有可能超过255，但是饱和度对值范围有要求，因此最终结果不会超过255。这样做的结果，就是这里的maxDist参数也必须小于256。如果你觉得这样不合理，可以进行修改。最后一步是用阈值函数创建一个二值图像。这个函数通常用于比较所有像素和某个阈值（第三个参数），并且在常规阈值化模式（cv::THRESH_BINARY）下，对所有大于阈值且大于0的像素赋值为预定的最大值（第四个参数）。这里我们使用相反的模式（cv::THRESH_BINARY_INV），把小于或等于阈值的像素赋值为预定的最大值。此外还有cv::THRESH_TOZERO_INV和cv::THRESH_TOZERO_INV模式，它们使大于或小于阈值的像素保持不变。

很多情况下，OpenCV函数都是一个不错的选择。你可以使用它快速地建立复杂程序，减少潜在的错误，而且程序的运行效率通常也比较高（得益于OpenCV项目参与者做的优化工作）。不过这样会执行很多的中间步骤，消耗更多内存。

3. 仿函数或函数对象

利用C++的操作符重载功能，我们可以创建一个其实例表现得像函数的类。它的原理是重载

operator()方法，因此调用类的处理方法就像调用一个纯粹的函数。这种类的实例被称为函数对象或者仿函数（functor）。一个仿函数通常包含一个完整构造函数，因此能够在创建后立即使用。例如，可以在ColorDetector类中添加下面的构造函数：

```
// 完整构造函数
ColorDetector(uchar blue, uchar green, uchar red,
              int maxDist=100): maxDist(maxDist) {

  // 目标颜色
  setTargetColor(blue, green, red);
}
```

很显然，前面定义的获取方法和设置方法仍然可以使用。可以这样定义仿函数方法：

```
cv::Mat operator()(const cv::Mat &image) {

  // 这里放检测颜色的代码
}
```

用仿函数方法检测指定的颜色，只需要用这样的代码片段：

```
ColorDetector colordetector(230,190,130,  // 颜色
                                    100); // 阈值
cv::Mat result= colordetector(image);     // 调用仿函数
```

可以看到，这里对颜色检测方法的调用类似于对某个函数的调用。事实上，colordetector变量可以当作一个函数名使用。

3.2.5　参阅

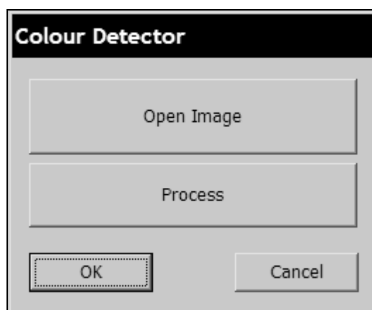
- A. Alexandrescu提出的“基于策略的类设计”是策略设计模式的一个变种，它把算法的选择放在编译时进行。
- Erich Gamma等人写的《设计模式：可复用面向对象软件的基础》（*Design Patterns: Elements of Reusable Object-Oriented Software*，Addison-Wesley于1994年出版）是这方面的一本经典著作。

3.3　用控制器设计模式实现功能模块间通信

在构建更复杂的程序时，你需要创建多个算法来协同工作，以实现一些高级功能。要合理地构建程序并让所有的类能互相通信，程序将会变得越来越复杂。因此在一个类中集中对程序进行控制，是非常有益的。这正是控制器设计模式背后的思想。控制器是一个特殊的对象，充当着程序中心的角色，本节将详细讨论。

3.3.1　准备工作

使用你喜欢的IDE建立一个简单的对话框程序，并创建两个按钮，一个用来选择图像，另一个用来启动处理，如下图所示：



这里我们使用上节的`ColorDetector`类。

3.3.2　如何实现

`Controller`类的首要任务是创建执行程序所需的类。这里只有一个类，但更复杂的程序就会有多个类。另外需要有两个成员变量，作为对输入和输出结果的引用：

```
class ColorDetectController {

  private:

   // 包含算法的类
   ColorDetector *cdetect;

   cv::Mat image; // 被处理的图像
   cv::Mat result;// 结果图像

  public:

   ColorDetectController() {

        //建立程序
        cdetect= new ColorDetector();
   }
```

这里我们采用动态地分配类的方式，也可以简单地声明类的变量。接着需要定义用于控制程序的所有设置方法和获取方法：

```
// 设置颜色差距的阈值
void setColorDistanceThreshold(int distance) {

   cdetect->setColorDistanceThreshold(distance);
```

```
}

// 取得颜色差距的阈值
int getColorDistanceThreshold() const {

   return cdetect->getColorDistanceThreshold();
}

// 设置被检测的颜色
void setTargetColor(unsigned char red,
   unsigned char green, unsigned char blue) {
        cdetect->setTargetColor(blue,green,red);
}

// 取得被检测的颜色
void getTargetColor(unsigned char &red,
   unsigned char &green, unsigned char &blue) const {

   cv::Vec3b color= cdetect->getTargetColor();

   red= color[2];
   green= color[1];
   blue= color[0];
}

// 设置输入图像。从文件中读取它
bool setInputImage(std::string filename) {

   image= cv::imread(filename);

   return !image.empty();
}

// 返回当前的输入图像
const cv::Mat getInputImage() const {

   return image;
}
```

你还需要一个启动处理过程的方法，供以后调用：

```
// 执行图像处理
void process() {

   result= cdetect->process(image);
}
```

此外，你还需要一个方法来获取处理结果：

```
// 返回最后处理的结果图像
const cv::Mat getLastResult() const {

   return result;
}
```

最后，非常重要的一步是在程序结束（释放Controller类）时做清理：

```
// 删除由控制器创建的对象
~ColorDetectController() {

   delete cdetect; // 释放动态分配的类实例的内存
}
```

3.3.3 实现原理

利用前面提到的Controller类，开发人员可以很容易地构建执行算法的程序接口，既不需要理解类与类是如何连接的，也不需要知道为了让所有类正确运行需要调用哪个类的哪个方法。所有这些工作都由Controller类完成。你唯一需要做的，就是创建一个Controller类的实例。

要部署算法，必须在Controller类中定义设置方法和获取方法。通常它们只是简单地调用相关类中对应的方法。这个例子只用了一个算法类，但实际开发中通常会包含多个类。因此，Controller的作用就是将请求重新定向到相关的类（在面向对象的编程中，这种机制称为委托）。控制器模式的另一个目的是简化程序中类的接口。作为这种简化的例子，setTargetColor和getTargetColor方法都使用uchar来设置和获取有关颜色，这样可以让程序开发者不需要掌握cv::Vec3b类的全部细节。

在一些情况下，控制器也可以准备应用开发者提供的数据。例如setInputImage方法会根据给定的文件名，把图像装载进内存。根据装载操作成功与否，方法返回true或false（这时也会发出一个异常提示）。

最后，用process方法运行算法。但是这个方法并不返回结果。要想得到最新的处理结果，必须调用另一个方法。

现在我们只需要在对话框类（这里称为colordetect）中添加一个ColorDetect-Controller成员变量，即可创建一个最基本的使用了控制器的对话框程序了。如果使用MS Visual Studio，则Open button按钮在MFC对话框中的消息处理函数如下：

```
// "Open"按钮的消息处理函数
void OnOpen()
{
    // 选择bmp或jpg类型文件的MFC对话框
    CFileDialog dlg(TRUE, _T("*.bmp"), NULL,
     OFN_FILEMUSTEXIST|OFN_PATHMUSTEXIST|OFN_HIDEREADONLY,
     _T("image files (*.bmp; *.jpg)
         |*.bmp;*.jpg|All Files (*.*)|*.*||"),NULL);

    dlg.m_ofn.lpstrTitle= _T("Open Image");

    // 如果选中了一个文件名
    if (dlg.DoModal() == IDOK) {
```

```
        // 取得选定文件名的完整路径
        std::string filename= dlg.GetPathName();

        // 设置并显示输入的图像
        colordetect.setInputImage(filename);
        cv::imshow("Input Image",colordetect.getInputImage());
      }
}
```

第二个按钮执行Process方法并显示结果：

```
//“Process”按钮的回调方法
void OnProcess()
{
   // 这里目标颜色采用硬编码
   colordetect.setTargetColor(130,190,230);
   // 处理输入图像并显示结果
   colordetect.process();
   cv::imshow("Output Result",colordetect.getLastResult());
}
```

如果程序更加复杂，显然会有更多的对话框来让用户选择算法的参数。

3.3.4　扩展阅读

在开发应用程序时一定要花时间规划一下架构，以方便日后维护和升级。有很多现成的架构模式，对优化架构很有帮助。

MVC架构

模型–视图–控制器（MVC）架构的目的是生成一个能把程序逻辑与用户接口清晰地隔离的程序，正如它的名称所示，MVC模式主要包括三个组件。

模型存放与应用程序有关的信息。它控制着程序处理的所有数据，当产生新数据时，它会通知控制器（通常是异步地），然后控制器通知视图显示新的结果。模型通常会整合多个算法，可能是通过策略模式实现的。所有这些算法是模型的一部分。

视图相当于用户接口。它由不同的窗口组成，窗口可向用户展示数据并且允许用户与程序交互。视图的功能之一，就是把用户发出的命令发送给控制器。当有新数据时，视图会刷新以显示新的信息。

控制器是连接视图和模型的模块。它从视图接收请求，并转发给模型中对应的方法。模型状态变化时会通知控制器，然后控制器通知视图进行刷新，以显示新的信息。

在MVC架构下，用户接口调用控制器的方法。接口不包含任何程序数据，也不实现任何程

序逻辑，因此接口很容易进行替换。用户界面设计者不需要理解程序的功能。反之，修改程序逻辑时也不需要通知界面设计者。

3.4 转换颜色表示法

前面几节介绍了如何把算法封装进类，这样我们就能通过简单的接口使用算法。通过封装，可以做到在修改一个算法的实现方法时不影响使用它的类。下一节将阐明这个原理，即为了使用另一个色彩空间而修改ColorDetector类中的算法。本节也会介绍OpenCV中的颜色转换。

3.4.1 准备工作

RGB色彩空间的基础是对叠加型三原色（红、绿、蓝）的应用。之所以选择它们，是因为把它们组合起来后可以产生色域很宽的各种颜色。实际上，人类的视觉系统也是基于对三原色的感知，因为视锥细胞的灵敏度位于红绿蓝的光谱周围。这通常是数字成像中默认的色彩空间，因为这就是人类看数字图像的方式。捕捉到的光线穿过红绿蓝三种滤波器，并且在数字图像中会对红绿蓝三个通道做校正，当三种颜色强度相同时就会取得灰度，即从黑色（0, 0, 0）到白色（255, 255, 255）。

但是，利用RGB色彩空间计算颜色之间的差距，并不是衡量两个颜色相似度的最好方式。实际上RGB并不是感知均匀的色彩空间。就是说，两种具有一定差距的颜色可能看起来非常接近，而另外两种具有同样差距的颜色看起来却差别很大。

为解决这个问题，引入了一些具有感知均匀特性的颜色表示法。CIE L*a*b*就是一种这样的颜色模型。把图像转换到这种表示法后，我们就可以真正地使用图像像素与目标颜色之间的欧几里德距离，来度量颜色之间的视觉相似度。本节介绍如何修改前面的程序，以适应CIE L*a*b*。

3.4.2 如何实现

使用OpenCV函数cv::cvtColor可以很容易地转换图像的色彩空间。在过程方法开始时把输入图像转换成CIE L*a*b*颜色空间：

```
cv::Mat ColorDetector::process(const cv::Mat &image) {

      // 必要时重新分配二值图像
      // 与输入图像的尺寸相同，但用单通道
      result.create(image.rows,image.cols,CV_8U);

      // 转换成Lab色彩空间
      cv::cvtColor(image, converted, CV_BGR2Lab);

      // 取得转换图像的迭代器
```

```
    cv::Mat_<cv::Vec3b>::iterator it=
                converted.begin<cv::Vec3b>();
    cv::Mat_<cv::Vec3b>::iterator itend=
                converted.end<cv::Vec3b>();
    // 取得输出图像的迭代器
    cv::Mat_<uchar>::iterator itout= result.begin<uchar>();

    // 针对每个像素
    for ( ; it!= itend; ++it, ++itout) {
    ...
```

转换后的变量包含颜色转换后的图像，被定义为类ColorDetector的一个属性：

```
class ColorDetector {

  private:
    // 颜色转换后的图像
    cv::Mat converted;
```

输入的目标颜色也需要进行转换，通过创建一个只有单个像素的临时图像，可以实现这种转换。注意，需要让函数保持与前面几节中一样的签名，即用户提供的目标颜色仍然是RGB格式的：

```
// 设置需要检测的颜色
void setTargetColor(unsigned char red,
      unsigned char green, unsigned char blue) {

    // 临时的单像素图像
    cv::Mat tmp(1,1,CV_8UC3);
    tmp.at<cv::Vec3b>(0,0)= cv::Vec3b(blue, green, red);

    // 目标颜色转换成Lab色彩空间
    cv::cvtColor(tmp, tmp, CV_BGR2Lab);

    target= tmp.at<cv::Vec3b>(0,0);
}
```

在上一节的程序中使用这个修改过的类，它就会在检测符合目标颜色的像素时，使用CIE L*a*b*颜色模型。

3.4.3　实现原理

在将图像从一个色彩空间转换到另一个色彩空间时，会在每个输入像素上做一个线性或非线性的转换，以得到输出像素。输出图像的像素类型与输入图像是一致的。即使你经常使用8位像素，也可以用浮点数图像（这时通常假定像素值的范围是0~1.0）或整数图像（像素值范围通常是0~65535）进行颜色转换。但是，实际的像素值范围取决于指定的色彩空间和目标图像的类型。例如，CIE L*a*b*色彩空间中的L通道表示每个像素的亮度，范围是0~100；在使用8位图像时，它的范围就会调整为0~255。a和b通道表示色度组件。这些通道包含了像素的颜色信息，与亮度无关。它们的值的范围是–127~127；对于8位图像，为了适合0到255的区间，每个值会加上128。

但是要注意，进行8位的颜色转换时会产生舍入误差，因此转换过程并不是完全可逆的。

大多数常用的色彩空间都是可以转换的。你只需要在OpenCV函数中指定正确的色彩空间转换代码（对于CIE L*a*b*，代码为`CV_BGR2Lab`），其中就有YCrCb，它是在JPEG压缩中使用的色彩空间。把色彩空间从BGR转换成YCrCb的代码为`CV_BGR2YCrCb`。注意，所有涉及三原色（红、绿、蓝）的转换过程，都可以用RGB和BGR的次序。

CIE L*u*v*是另一种感知均匀的色彩空间。从BGR转换成CIE L*u*v*，可使用代码`CV_BGR2Luv`。对于亮度通道，L*a*b*和L*u*v*都使用同样的转换公式，但对色度通道则使用不同的表示法。另外，为了实现视觉感知上的均匀，这两种色彩空间都扭曲了RGB的颜色范围，所以这些转换过程都是非线性的（因此它们的计算量很大）。

此外还有CIE XYZ色彩空间（用代码`CV_BGR2XYZ`表示）。它是一种标准的色彩空间，用与设备无关的方式表示任何可见颜色。在L*a*b*和L*u*v*色彩空间的计算中，用XYZ色彩空间作为一种中间表示法。RGB与XYZ之间的转换是线性的。还有一点非常有趣，就是Y通道对应着图像的灰度版本。

HSV和HLS这两种色彩空间很有意思，它们把颜色分解成加值的色调和饱和度组件或亮度组件。人类用这种方式来描述颜色会更加自然。

你可以把彩色图像转换成灰度图像，输出是一个单通道图像：

```
cv::cvtColor(color, gray, CV_BGR2Gray);
```

也可以进行反向的转换，但是那样得到彩色图像的三个通道是相同的，都是灰度图像中对应的值。

3.4.4　参阅

- 4.6节使用了HSV色彩空间来寻找图像中的目标。
- 关于色彩空间理论的参考资料有很多，其中有一套完整的资料：*The Structure and Properties of Color Spaces and the Representation of Color Images*（E. Dubois著，Morgan & Claypool，2009年出版）。

3.5　用色调、饱和度、亮度表示颜色

本章我们处理了图像的颜色，使用了不同的色彩空间，并且设法识别出图像中具有特定颜色的区域。例如，RGB是一种被广泛接受的色彩空间，被视为一种非常有效的表示法，用于在电子成像系统中采集和显示颜色。但是这种表示法并不非常直观，它并不符合人类对于颜色的感知方式。我们谈论颜色时，总会提到色彩、亮度或彩度（即表示该颜色是鲜艳的还是柔和的）。直觉

色彩空间基于色调、饱和度、亮度的概念，可以让人们用更直观的属性来描述颜色。本节我们把色调、饱和度、亮度作为描述颜色的方法，并探讨这些概念。

3.5.1　如何实现

上一节我们讲过，可用`cv::cvtColor`函数把BGR图像转换成直觉色彩空间。这里我们使用转换代码`CV_BGR2HSV`：

```
// 转换成HSV色彩空间
cv::Mat hsv;
cv::cvtColor(image, hsv, CV_BGR2HSV);
```

我们可以用代码`CV_HSV2BGR`把图像转换回BGR色彩空间。通过把图像的通道分割到三个独立的图像中，我们可以直观地看到每一种HSV组件。方法如下：

```
// 把三个通道分割进三个图像中
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// channels[0]是色调
// channels[1]是饱和度
// channels[2]是亮度
```

因为处理的是8位图像，OpenCV会把通道值的范围重新调节为0~255（色调除外，色调的范围被调节为0~180，下面会解释原因）。这个方法非常实用，因为我们可以把这几个通道作为灰度图像进行显示。城堡图的亮度通道显示如下：



该图像的饱和度通道显示如下：



最后是该图像的色调通道：



下一节会对这几个图像进行解释。

3.5.2　实现原理

之所以要引入直觉色彩空间的概念，是因为人类倾向于自然地组织各种颜色，而直觉色彩空间与这种方式相吻合。实际上，人类喜欢用色彩、彩度、亮度等直观的属性来描述颜色，而大多数直觉色彩空间正是基于这三种属性。色调（hue）表示主色。我们使用的颜色名称（例如绿色、黄色和红色）就对应了不同的色调值；饱和度（saturation）表示颜色的鲜艳程度。柔和的颜色饱和度较低，而彩虹的颜色饱和度很高；最后，亮度（brightness）是一个主观的属性，表示某种颜色的光亮程度。其他直觉色彩空间使用颜色明度（value）或颜色亮度（lightness）的概念描述有关颜色的强度。

人们利用这些颜色概念，尽可能地模拟人类对颜色的直观感知。因此，它们没有标准的定义。从字面上看，色调、饱和度、亮度都有多种不同的定义和公式。OpenCV所建议的两种直觉色彩空间的实现是HSV和HLS色彩空间。它们的转换公式略有不同，但是结果非常类似。

亮度成分可能是最容易解释的。在OpenCV对HSV的实现中，它被定义为三个BGR成分中的最大值，以非常简化的方式实现了亮度的概念。为了让定义更符合人类视觉系统，你应该使用L*a*b*或L*u*v*色彩空间的L通道。

OpenCV用一个公式来计算饱和度，该公式基于BGR组件的最小值和最大值：

$$s = \frac{\max(R,G,B) - \min(R,G,B)}{\max(R,G,B)}$$

其原理是：灰度颜色所包含的R、G和B成分是相等的，它相当于一种极不饱和的颜色，因此它的饱和度是0（饱和度是一个从0~1.0的值）。对于8位图像，饱和度被调节成一个从0到255的值，并且作为灰度图像显示的时候，较亮的区域对应的颜色具有较高的饱和度。例如在前面的饱和度图片中，水的蓝色比天空的柔和浅蓝色的饱和度高，这和我们的推断是一致的。根据定义，各种灰色阴影的饱和度都是0（因为它们的三种BGR组件是相等的）。在城堡的屋顶能看到这种现象，因为屋顶是深灰色石头组成的。最后，在饱和度图像中可以看到一些白色的斑点，所处位置对应原始图像中非常暗的区域。这是由饱和度的定义引起的。饱和度只计算BGR中最大值和最小值的相对差距，因此像（1, 0, 0）这样的组合就会得到饱和度1.0，尽管这个颜色看起来是黑色的。因此黑色区域中计算得到的饱和度是不可靠的，没有参考价值。

颜色的色调通常用0~360的角度来表示，其中红色是0度。对于8位图像，OpenCV把角度除以2，以适合单字节的存储范围。因此，每个色调值对应指定颜色的色彩，与亮度和饱和度无关。例如天空和水的色调是一样的，约为200度（强度100），对应色度为蓝色；背景树林的色调约为90度，对应色度为绿色。有一点要特别注意，如果颜色的饱和度很低，它计算出来的色调就不可靠。

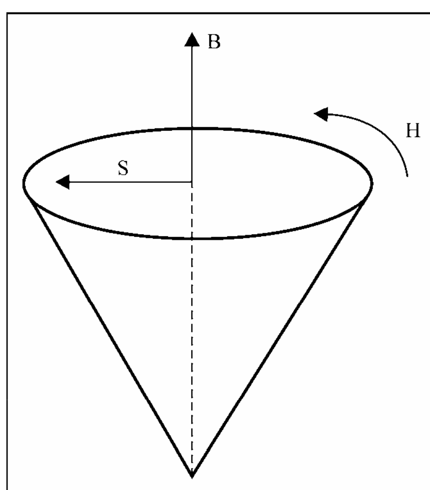
参见下页图（左），HSB色彩空间通常用一个圆锥体来表示，圆锥体内部每个点代表一种特定的颜色。角度位置表示颜色的色调，到中轴线的距离表示饱和度，高度表示亮度。圆锥体的顶点表示黑色，它的色调和饱和度是没有意义的。

使用HSV的值可以生成一些非常有趣的效果。一些用照片编辑软件生成的色彩特效，就是用这个色彩空间实现的。例如你可以修改一个图像，把它的所有像素设置为一个固定的亮度，但不改变色调和饱和度。可以这样实现：

```
// 转换成HSV色彩空间
cv::Mat hsv;
cv::cvtColor(image, hsv, CV_BGR2HSV);
// 分割3个通道到3个图像中
```

```
std::vector<cv::Mat> channels;
cv::split(hsv,channels);
// 所有像素的颜色亮度通道将变成255
channels[2]= 255;
// 重新合并通道
cv::merge(channels,hsv);
// 转换回BGR
cv::Mat newImage;
cv::cvtColor(hsv,newImage,CV_HSV2BGR);
```

得到的结果如下图（右）所示，看起来像是一幅绘画作品（可通过本书的图片集查看彩色图像）：





3.5.3　扩展阅读

在搜寻特定颜色的物体时，HSV色彩空间也是非常实用的。

颜色用于检测：肤色检测

在对特定物体做初步检测时，颜色信息非常有用。例如在辅助驾驶程序中检测路标，就要凭借标准路标的颜色快速地提取出可能是路标的信息。另一个例子是检测皮肤的颜色，检测到的皮肤区域可作为图像中有人存在的标志。在手势识别中经常使用这个方法，用肤色检测来确定手的位置。

通常来说，为了用颜色来检测目标，首先需要收集一个存储有大量图像样本的数据库，每个样本中包含从不同观察条件下捕捉到的目标，作为定义分类器的参数。你还需要选择一种用于分类的颜色表示法。肤色检测领域的大量研究已经表明，来自不同人种的人群的皮肤颜色，可以在色调-饱和度色彩空间中很好地归类。因此，在后面的图像中，我们将只使用色调和饱和度值来

识别肤色（可通过本书的图片集查看彩色图像，另见彩图 4 ）：



因此我们定义了一个基于数值区间（最小和最大色调、最小和最大饱和度）的函数，把图像中的像素分为皮肤和非皮肤两类：

```
void detectHScolor(const cv::Mat& image, // 输入图像
  double minHue, double maxHue,  // 色调区间
  double minSat, double maxSat,  // 饱和度区间
  cv::Mat& mask) {               // 输出掩码

  // 转换到HSV空间
  cv::Mat hsv;
  cv::cvtColor(image, hsv, CV_BGR2HSV);

  // 分割3个通道，并存进3个图像
  std::vector<cv::Mat> channels;
  cv::split(hsv, channels);
  // channels[0] 是色调
  // channels[1] 是饱和度
  // channels[2] 是亮度

  // 色调掩码
  cv::Mat mask1; // 小于maxHue
  cv::threshold(channels[0], mask1, maxHue, 255,
  cv::THRESH_BINARY_INV);
  cv::Mat mask2; // 大于minHue
  cv::threshold(channels[0], mask2, minHue, 255,
  cv::THRESH_BINARY);

  cv::Mat hueMask; // 色调掩码
  if (minHue < maxHue)
      hueMask = mask1 & mask2;
  else // 如果区间穿越0度中轴线
      hueMask = mask1 | mask2;

  // 饱和度掩码
  // 小于maxSat
  cv::threshold(channels[1], mask1, maxSat, 255,
  cv::THRESH_BINARY_INV);
```

```
  // 大于minSat
  cv::threshold(channels[1], mask2, minSat, 255,
  cv::THRESH_BINARY);

  cv::Mat satMask; // 饱和度掩码
  satMask = mask1 & mask2;

  // 组合掩码
  mask = hueMask&satMask;
}
```

在处理时有了大量的皮肤（以及非皮肤）样本，我们可以使用概率方法比较在皮肤样本中和在非皮肤样本中发现指定颜色的可能性。此处，我们依据经验定义一个合理的色调–饱和度区间，用于这里的测试图像（记住，8位版本的色调在0到180之间，饱和度在0到255之间）：

```
// 检测肤色
cv::Mat mask;
detectHScolor(image,
      160, 10, // 色调从320度到20度
      25, 166, // 饱和度从~0.1到0.65
      mask);

// 显示使用掩码后的图像
cv::Mat detected(image.size(), CV_8UC3, cv::Scalar(0, 0, 0));
image.copyTo(detected, mask);
```

得到下面的检测图像（另见彩图5）：



注意，为了简化，我们在检测中没有考虑颜色的饱和度。实际上，排除较高饱和度的颜色可以降低把明亮的淡红色误认为皮肤的可能性。显然，对皮肤颜色进行可靠和准确地检测需要更加精确的基于大批量皮肤样本的分析。并且对不同的图像进行检测，很难保证都有好的效果。这是因为摄影时影响彩色再现的因素很多，如白平衡和光照条件等。尽管如此，用本章的方法，只使用色调信息做初步的检测，我们也能得到一个合理的结果。

第 4 章

用直方图统计像素

本章包括以下内容：

- 计算图像直方图；
- 利用查找表修改图像外观；
- 直方图均衡化；
- 反向投影直方图检测特定图像内容；
- 均值平移算法查找目标；
- 比较直方图搜索相似图像；
- 用积分图像统计像素。

4.1 简介

图像是由不同数值（颜色）的像素构成的，像素值在整幅图像中的分布情况是该图像的一个重要属性。本章介绍图像直方图的概念，你将学会计算直方图、用直方图修改图像的外观，还可以用直方图来标识图像的内容，在图像中检测特定物体或纹理。本章将讲解其中的部分技术。

4.2 计算图像直方图

图像由像素构成，每个像素有不同的数值。例如，在单通道灰度图像中，每个像素都有一个0（黑色）~255（白色）的数值。对于每个灰度，都有不同数量的像素分布在图像内，具体取决于图片内容。

*直方图*是一个简单的表格，表示一个图像（有时是一组图像）中具有某个值的像素的数量。因此灰度图像的直方图有256个项目，也叫箱子（bin）。0号箱子提供值为0的像素的数量，1号箱子提供值为1的像素的数量，等等。很明显，如果把直方图的所有箱子进行累加，得到的结果就是像素的总数。你也可以把直方图归一化，即所有箱子的累加和等于1。这时每个箱子的数值表示对应的像素数量占总数的百分比。

4.2.1 准备工作

本章的前三节会用到这个图像：



4.2.2 如何实现

要在OpenCV中计算直方图，可简单地调用`cv::calcHist`函数。这是一个通用的直方图计算函数，可处理包含任何值类型和范围的多通道图像。为了简化，这里我们指定一个专门用于处理单通道灰度图像的类。`cv::calcHist`函数非常灵活，在处理其他类型的图像时都可以直接使用它。下一节会解释它的每个参数。

现在，这个专用的类如下所示：

```
// 创建灰度图像的直方图
class Histogram1D {

  private:

    int histSize[1];         // 直方图中箱子的数量
    float hranges[2];        // 值范围
    const float* ranges[1];  // 值范围的指针
    int channels[1];         // 要检查的通道数量

  public:

  Histogram1D() {

    // 准备一维直方图的默认参数
    histSize[0]= 256;   // 256个箱子
    hranges[0]= 0.0;    // 从0开始（含）
    hranges[1]= 256.0;  // 到256（不含）
    ranges[0]= hranges;
    channels[0]= 0;     // 先关注通道0
  }
```

定义好成员变量后，就可以用下面的方法计算灰度直方图了：

```
// 计算一维直方图
cv::Mat getHistogram(const cv::Mat &image) {

  cv::Mat hist;

  // 计算直方图
  cv::calcHist(&image,
    1,          // 仅为一个图像的直方图
    channels,   // 使用的通道
    cv::Mat(),  // 不使用掩码
    hist,       // 作为结果的直方图
    1,          // 这是一维的直方图
    histSize,   // 箱子数量
    ranges      // 像素值的范围
  );

  return hist;
}
```

现在程序只需要打开一个图像，创建一个`Histogram1D`实例，然后调用`getHistogram`方法：

```
// 读取输入的图像
cv::Mat image= cv::imread("group.jpg",
                          0); // 以黑白方式打开

// 直方图对象
Histogram1D h;

// 计算直方图
cv::Mat histo= h.getHistogram(image);
```

这里的`histo`对象是一个一维数组，包含256个项目。因此只需遍历这个数组，就可以读取每个箱子：

```
// 循环遍历每个箱子
for (int i=0; i<256; i++)
   cout << "Value " << i << " = " <<
                histo.at<float>(i) << endl;
```

使用本章开始时使用的图像，部分显示的值如下所示：

```
...
Value 7 = 159
Value 8 = 208
Value 9 = 271
Value 10 = 288
Value 11 = 340
Value 12 = 418
Value 13 = 432
Value 14 = 472
Value 15 = 525
...
```

显然，很难从这一系列数值中得到任何直观的意义。因此比较实用的做法是以函数的方式显示直方图，例如用柱状图。用下面的几个方法可创建这种图形：

```
// 计算一维直方图，并返回它的图像
cv::Mat getHistogramImage(const cv::Mat &image,
                             int zoom=1){

  // 首先计算直方图
  cv::Mat hist= getHistogram(image);

  // 创建图像
  return getImageOfHistogram(hist, zoom);
}

// 创建表示一个直方图的图像（静态方法）
static cv::Mat getImageOfHistogram
                  (const cv::Mat &hist, int zoom) {

  // 取得箱子值的最大值和最小值
  double maxVal = 0;
  double minVal = 0;
  cv::minMaxLoc(hist, &minVal, &maxVal, 0, 0);

  // 取得直方图的大小
  int histSize = hist.rows;

  // 用于显示直方图的方形图像
  cv::Mat histImg(histSize*zoom,
                   histSize*zoom, CV_8U, cv::Scalar(255));

  // 设置最高点为90%（即图像高度）的箱子个数
  int hpt = static_cast<int>(0.9*histSize);

  // 为每个箱子画垂直线
  for (int h = 0; h < histSize; h++) {

    float binVal = hist.at<float>(h);
    if (binVal>0) {
      int intensity = static_cast<int>(binVal*hpt / maxVal);
      cv::line(histImg, cv::Point(h*zoom, histSize*zoom),
        cv::Point(h*zoom, (histSize - intensity)*zoom),
             cv::Scalar(0), zoom);
    }
  }

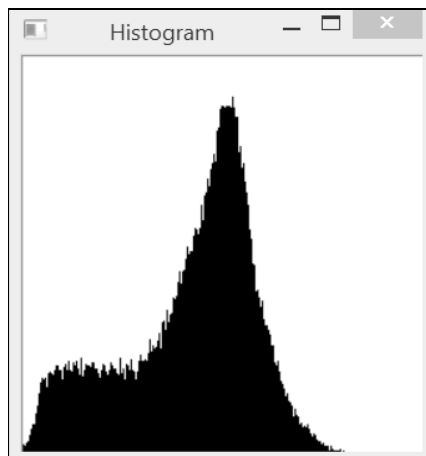
  return histImg;
}
```

使用getImageOfHistogram方法可以得到直方图图像。它用线条画成，以柱状图形式展现：

```
// 以图像形式显示直方图
cv::namedWindow("Histogram");
cv::imshow("Histogram",
```

```
        h.getHistogramImage(image));
```

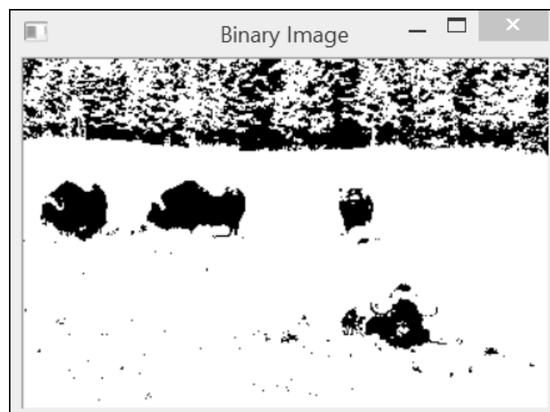
得到的结果如下图：



从上面图形化的直方图可以看出，在中等灰度值处有一个大的尖峰，并且比中等值更黑的像素数量很大。巧合的是，这两部分像素分别对应了图像的背景和前景。要验证这点，可以在两部分的汇合处进行阈值化处理。OpenCV中的`cv::threshold`函数可以实现这个功能。上一章介绍过，它是一个很实用的函数。我们取直方图中在升高为尖峰之前的最小值的位置（灰度值为`60`），对其进行阈值化处理，得到二值图像：

```
cv::Mat thresholded; // 输出二值图像
cv::threshold(image,thresholded,
            60,    // 阈值
            255,   // 对超过阈值的像素赋值
          cv::THRESH_BINARY); // 阈值化类型
```

得到的二值图像清晰的显示出背景/前景的分割情况：



4.2.3　实现原理

为了适应各种场景，cv::calcHist函数带有很多参数，如下：

```
void calcHist(const Mat* images, int nimages,
  const int* channels, InputArray mask, OutputArray hist,
  int dims, const int* histSize, const float** ranges,
  bool uniform=true, bool accumulate=false )
```

大多数情况下，直方图是单个的单通道或三通道图像。但也可以在这个函数中指定一个分布在多个图像上的多通道图像。这也是函数输入中用图像数组的原因。第六个参数dims指明了直方图的维数。例如1表示一维直方图。在分析多通道图像时，可以只把它的部分通道用于计算直方图，将需要处理的通道放在维数确定的数组channel中。在这个类的实现中只有一个通道，默认为0。直方图用每个维度上的箱子数量（即整数数组histSize）以及每个维度（由ranges数组提供，数组中每个元素又是一个二元素数组）上的最小值（含）和最大值（不含）来描述。你也可以定义一个不均匀的直方图，这时需要指定每个箱子的限值。

和很多OpenCV函数一样，可以使用掩码表示计算时用到的像素（所有掩码值为0的像素都不被使用）。此外还可以指定两个布尔值类型的附加参数。第一个表示是否采用均匀的直方图（默认为均匀）；第二个表示是否允许累加多个直方图计算的结果。如果第二个参数为“是”，那么图像中的像素数量会累加到输入直方图的当前值中。在计算一组图像的直方图时，就可以使用这个参数。

得到的直方图存储在cv::Mat的实例中。实际上，cv::Mat类可用于操作通用的N维矩阵。第2章讲过，cv::Mat类定义了适用于一维、二维和三维矩阵的at方法。正因为如此，我们才可以在getHistogramImage方法中用下面的代码访问一维直方图的每个箱子：

```
float binVal = hist.at<float>(h);
```

注意，直方图中的值被存储为float值。

4.2.4　扩展阅读

本节中的Histogram1D类简化了cv::calcHist函数，把它限定为只用于一维直方图。这对灰度图像是有用的，但是怎么处理彩色图像呢？

计算彩色图像的直方图

我们可以用同一个cv::calcHist函数计算多通道图像的直方图。例如，要计算彩色BGR图像的直方图，可以这样定义一个类：

```
class ColorHistogram {
```

```
private:

  int histSize[3];        // 每个维度的大小
  float hranges[2];       // 值的范围
  const float* ranges[3]; // 每个维度的范围
  int channels[3];        // 需要处理的通道

public:

ColorHistogram() {

  // 准备用于彩色图像的默认参数
  // 每个维度的大小和范围是相等的
  histSize[0]= histSize[1]= histSize[2]= 256;
  hranges[0]= 0.0;    // BGR范围为0~256
  hranges[1]= 256.0;
  ranges[0]= hranges; // 这个类中，
  ranges[1]= hranges; // 所有通道的范围都相等
  ranges[2]= hranges;
  channels[0]= 0;     // 三个通道
  channels[1]= 1;
  channels[2]= 2;
}
```

这里的直方图将会是三维的，因此需要为每个维度指定一个范围。本例中的BGR图像，三个通道的范围都是`[0,255]`。准备好参数后，就可以用下面的方法计算颜色直方图了：

```
// 计算直方图
cv::Mat getHistogram(const cv::Mat &image) {

  cv::Mat hist;

  // BGR颜色直方图
  hranges[0]= 0.0;    // BGR的范围
  hranges[1]= 256.0;
  channels[0]= 0;     // 三个通道
  channels[1]= 1;
  channels[2]= 2;

  // 计算直方图
  cv::calcHist(&image,
    1,           // 单个图像的直方图
    channels,    // 用到的通道
    cv::Mat(),   // 不使用掩码
    hist,        // 得到的直方图
    3,           // 这是一个三维直方图
    histSize,    // 箱子数量
    ranges       // 像素值的范围
  );

  return hist;
}
```

上述方法返回一个三维的cv::Mat实例。如果选用含有256个箱子的直方图，这个矩阵就有(256)^3个元素，表示超过1600万个项目。在很多程序中，最好在计算直方图时减少箱子的数量。也可以使用数据结构cv::SparseMat表示大型稀疏矩阵（即非零元素非常稀少的矩阵），而不会消耗过多的内存。cv::calcHist函数具有返回这种矩阵的版本，因此只需要简单地修改一下前面的方法，即可使用cv::SparseMatrix：

```
// 计算直方图
cv::SparseMat getSparseHistogram(const cv::Mat &image) {

  cv::SparseMat hist(3,          // 维数
                  histSize,    // 每个维度的大小
               CV_32F);

  // BGR颜色直方图
  hranges[0]= 0.0;    // BGR范围
  hranges[1]= 256.0;
  channels[0]= 0;     // 三个通道
  channels[1]= 1;
  channels[2]= 2;

  // 计算直方图
  cv::calcHist(&image,
    1,          // 单个图像的直方图
    channels,   // 用到的通道
    cv::Mat(),  // 不使用掩码
    hist,       // 得到的直方图
    3,          // 这是三维直方图
    histSize,   // 箱子数量
    ranges      // 像素值的范围
  );

  return hist;
}
```

显然，我们也可以通过显示独立的R、G和B通道的直方图说明图像中颜色的分布情况。

4.2.5 参阅

- ❑ 4.5节使用颜色直方图来检测特定的图像内容。

4.3 利用查找表修改图像外观

图像直方图为我们提供了利用现有像素强度值进行场景渲染的方法。通过分析图像中像素值的分布情况，你可以利用这个信息来修改图像，甚至改进图像质量。本节解释如何用一个简单的映射函数（我们称为查找表），来修改图像的像素值。我们即将看到，查找表通常根据直方分布图来定义。

4.3.1　如何实现

查找表是个一对一（或多对一）的函数，定义了如何把像素值转换成新的值。它是一个一维数组，对于规则的灰度图像，它包含256个项目。利用查找表的项目`i`，可得到对应灰度级的新强度值，如下所示：

```
newIntensity= lookup[oldIntensity];
```

OpenCV中的`cv::LUT`函数在图像上应用查找表，可生成一个新的图像。我们可以在`Histogram1D`类中加入这个函数：

```
static cv::Mat applyLookUp(
 const cv::Mat& image,    // 输入图像
 const cv::Mat& lookup) { // uchar类型的1 × 256数组

   // 输出图像
   cv::Mat result;

   // 应用查找表
   cv::LUT(image,lookup,result);

   return result;
}
```

4.3.2　实现原理

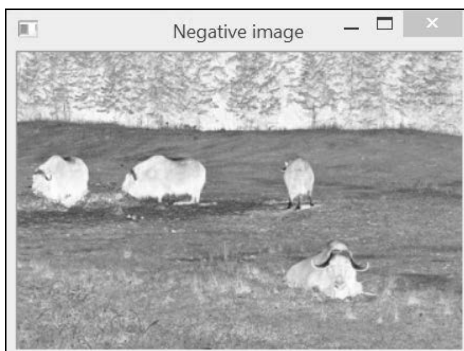
在图像上应用查找表后会得到一个新图像，新图像的像素强度值被修改为查找表中规定的值。下面是一个简单的转换过程：

```
// 创建一个图像反转的查找表
int dim(256);
cv::Mat lut(1, // 一维
   &dim,       // 256个项目
   CV_8U);     // uchar类型

for (int i=0; i<256; i++) {

   lut.at<uchar>(i)= 255-i;
}
```

这个转换过程对像素强度进行简单的反转，即强度0变成255、1变成254，等等。对图像应用这种查找表后，会生成原始图像的反向图像。使用上节的图像，得到结果如下：



4.3.3 扩展阅读

对于需要更换全部像素强度值的程序，都可以使用查找表。但是这个转换过程必须是针对整幅图像的。也就是说，一个强度值对应的全部像素都必须使用同一种转换方法。

1. 伸展直方图以提高图像对比度

定义一个修改原始图像直方图的查找表，可以提高图像的对比度。例如，观察第一节的图像直方图很容易发现：整个可用的强度值范围并没有被完全利用（特别是图像中比较亮的强度值并没有被利用）。我们可以通过伸展直方图来生成一个对比度更高的图像。为此要使用一个百分比阈值，表示伸展后图像的黑色和白色像素的百分比。

我们必须在强度值中找到一个最小值（imin）和最大值（imax），使得所要求的最小的像素数量高于阈值指定的百分比。然后重新映射强度值，使imin的值变成强度值0，imax的值变成强度值255。两者之间的强度值i可以简单地做线性重新映射，如下：

```
255.0*(i-imin)/(imax-imin);
```

因此，完整的图像伸展方法如下所示：

```
cv::Mat stretch(const cv::Mat &image, int minValue=0) {

   // 首先计算直方图
   cv::Mat hist= getHistogram(image);

   // 找到直方图的左边限值
   int imin= 0;
   for( ; imin < histSize[0]; imin++ ) {
      // 忽略数量少于minValue项目的箱子
      if (hist.at<float>(imin) > minValue)
         break;
   }

   // 找到直方图的右边限值
```

```
int imax= histSize[0]-1;
for( ; imax >= 0; imax-- ) {

   // 忽略数量少于minValue的箱子
   if (hist.at<float>(imax) > minValue)
      break;
}

// 创建查找表
int dim(256);
cv::Mat lookup(1,    // 一维
      &dim,          // 256个项目
      CV_8U);        // uchar类型

// 构建查找表
for (int i=0; i<256; i++) {

   // 在imin和imax之间伸展
   if (i < imin) lookup.at<uchar>(i)= 0;
   else if (i > imax) lookup.at<uchar>(i)= 255;
   // 线性映射
else lookup.at<uchar>(i)=
  cvRound(255.0*(i-imin)/(imax-imin));
}

// 应用查找表
cv::Mat result;
result= applyLookUp(image,lookup);

return result;
}
```

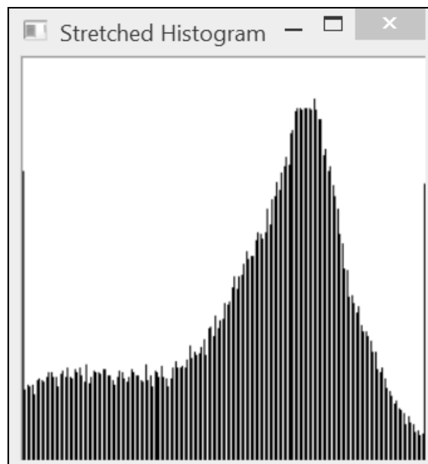
在计算完毕后，记得调用applyLookUp方法。另外根据经验，最好不要仅仅忽略值为0的箱子，还要忽略数量小到可以忽略不计的箱子。例如，小于某个特定的数值的箱子（这里用minValue表示）。这个方法的调用方式为：

```
// 把1%的像素设为黑色，1%的设为白色
cv::Mat streteched = h.stretch(image,0.01f);
```

伸展图像的结果如下：



扩展过的直方图如下：



2. 在彩色图像上应用查找表

第2章中我们定义了一个减色函数，通过修改图像中的BGR值减少可能的颜色数量。当时的实现方法是循环遍历图像中的像素，并对每个像素应用减色函数。实际上，更高效的做法是预先计算好所有的减色值，然后用查找表修改每个像素。利用本节的方法，这很容易实现。下面是新的减色函数：

```
void colorReduce(cv::Mat &image, int div=64) {

    // 创建一维查找表
    cv::Mat lookup(1,256,CV_8U);

    // 定义减色查找表的值
    for (int i=0; i<256; i++)
      lookup.at<uchar>(i)= i/div*div + div/2;

    // 对每个通道应用查找表
    cv::LUT(image,lookup,image);
}
```

这种减色方案之所以能在这里正确应用，是因为在多通道图像上应用一维查找表时，同一个查找表会独立地应用在所有通道上。如果查找表超过一个维度，那么它和所用图像的通道数必须相同。

4.3.4 参阅

- 下一节将展示另一种增强图像对比度的方法。

4.4　直方图均衡化

上节我们介绍了一种增强图像对比度的方法。即通过伸展直方图，使它布满可用强度值的全部范围。这方法确实可以简单有效地改进图像质量。但很多时候，图像的视觉缺陷并不是它使用的强度值范围太窄，而是由于部分强度值的使用频率比其他强度值要高得多。4.1节显示的直方图就是此类现象的一个很好的例子。中等灰度的强度值非常多，而较暗和较亮的像素值则非常稀少。通常认为，对所有可用像素强度值都均衡使用，才是一副高质量的图像。这正是直方图均衡化这一概念背后的思想，也就是让图像的直方图尽可能地平稳。

4.4.1　如何实现

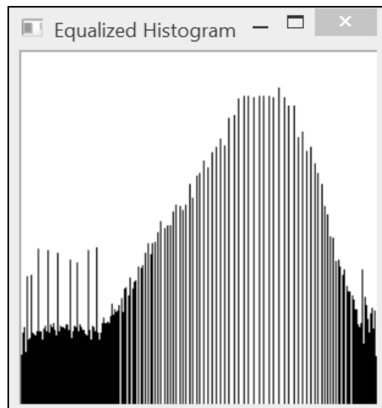
OpenCV提供了一个易用的函数，用于直方图均衡化处理。这个函数的调用方式为：

```
cv::equalizeHist(image,result);
```

对图像应用该函数后，得到的结果如下：



均衡化后图像的直方图如下：



当然，因为查找表是针对整幅图像的多对一的转换过程，所以直方图是不能做到完全平稳的。但是可以看出，直方图的一般分布情况已经比原来均衡多了。

4.4.2　实现原理

在一个完全均衡的直方图中，所有箱子所包含的像素数量是相等的。这意味着50%像素的强度值小于`128`，25%像素的强度值小于`64`，依此类推。这个现象可以用一条规则来表示：*p*%像素的强度值必须小于或等于255**p*%。这条规则用于直方图均衡化处理，表示强度值`i`的映像对应强度值小于`i`的像素所占的百分比。因此可以用下面的语句构建所需的查找表：

```
lookup.at<uchar>(i)=
        static_cast<uchar>(255.0*p[i]/image.total());
```


这里`p[i]`是强度值小于或等于`i`的像素数量，通常称为*累计直方图*。这种直方图包含小于或等于指定强度值的像素数量，而非仅仅包含等于指定强度值的像素数量。前面说过`image.total()`返回图像的像素总数，因此`p[i]/image.total()`就是像素数量的百分比。

通常直方图均衡化会极大地改进图像外观，但是改进的效果会因图像可视内容不同而出现差异。

4.5　反向投影直方图检测特定图像内容

直方图是图像内容的一个重要特性。如果图像的某个区域含有特定的纹理或物体，这个区域的直方图就可以看作一个函数，该函数返回某个像素属于这个特殊纹理或物体的概率。本节介绍如何运用*直方图反向投影*的概念方便地检测特定的图像内容。

4.5.1　如何实现

假设你希望在某个图像中检测出特定的内容（例如，在下图中间检测出天空中的云彩），首先要做的就是选择一个包含所需样本的兴趣区域。下图中，该区域就是绘制的矩形内部：



在程序中，用下面的方法可以得到这个兴趣区域：

```
cv::Mat imageROI;
imageROI= image(cv::Rect(216,33,24,30)); // 云彩区域
```

接着提取该兴趣区域（ROI）的直方图。使用4.1节的Histogram1D类，将很容易获得该直方图：

```
Histogram1D h;
cv::Mat hist= h.getHistogram(imageROI);
```

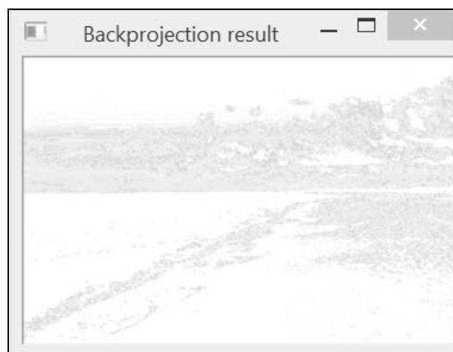
通过归一化直方图，我们可得到一个函数。由此可得到特定强度值的像素属于这个区域的概率：

```
cv::normalize(histogram,histogram,1.0);
```

反向投影直方图的过程包括：从归一化后的直方图中读取概率值并把输入图像中的每个像素替换成与之对应的概率值。OpenCV中有一个函数可完成此任务：

```
cv::calcBackProject(&image,
          1,           // 一个图像
          channels,    // 用到的通道，取决于直方图的维度
          histogram,   // 需要反向投影的直方图
          result,      // 反向投影得到的结果
          ranges,      // 值的范围
          255.0        // 选用的换算系数
          // 把概率值从1映射到255
    );
```

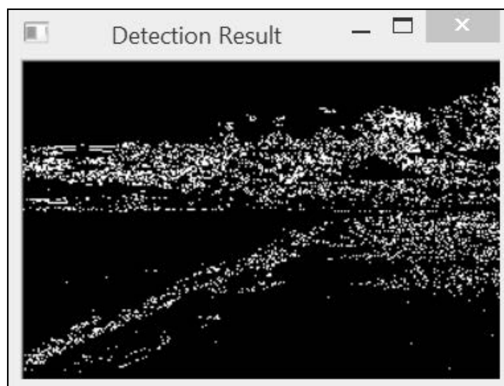
得到的结果就是下面的概率分布图，属于该区域的概率从亮（低概率）到暗（高概率）：



如果对此图做阈值化处理，就得到最有可能是“云彩”的像素：

```
cv::threshold(result, result, threshold,
                      255, cv::THRESH_BINARY);
```

得到的结果如下：



4.5.2 实现原理

前面的结果并不令人满意。因为除了云彩，其他区域也被错误地检测到了。这个概率函数是从一个简单的灰度直方图提取的，理解这点很重要。其他很多像素的强度值与云彩像素的强度值是相同的，在对直方图进行反向投影时会用相同的概率值替换具有相同强度值的像素。有一种方案可提高检测效果，就是使用色彩信息。要实现这点，需改变对cv::calBackProject的调用方式。

cv::calBackProject函数和cv::calcHist有些类似。第一个参数指明输入的图像，接着需要指明使用的通道数量。这里传递给函数的直方图是一个输入参数，它的维度数要与通道列表数组的维度数一致。与cv::calcHist函数一样，这里的ranges参数用数组形式指定了输入直方图的箱子边界。该数组以浮点数组为元素，每个数组元素表示一个通道的取值范围（最小值和最大值）。输出结果是一个图像，即计算得到的概率分布图。由于每个像素已经被替换成直方图中对应箱子处的值，因此输出图像的值范围是0.0~1.0（假定输入的是归一化直方图）。最后一个参数是换算系数，可用来重新缩放这些值。

4.5.3 扩展阅读

现在我们来学习如何在直方图反向映射算法中使用彩色信息。

反向映射颜色直方图

多维度直方图也可以在图像上进行反向映射。我们定义一个封装反向映射过程的类，首先定义必需的参数并初始化：

```
class ContentFinder {

  private:
```

```
// 直方图参数
float hranges[2];
 const float* ranges[3];
 int channels[3];

float threshold;         // 判断阈值
cv::Mat histogram;       // 输入直方图

public:

ContentFinder() : threshold(0.1f) {

  // 本类中，所有通道的范围相同
  ranges[0]= hranges;
  ranges[1]= hranges;
  ranges[2]= hranges;
}
```

接下来定义一个阈值参数，用于创建显示检测结果的二值分布图。如果这个参数设为负数，就会返回原始的概率分布图。参见以下代码：

```
// 设置直方图的阈值[0,1]
void setThreshold(float t) {

   threshold= t;
}

// 取得阈值
float getThreshold() {

   return threshold;
}
```

输入的直方图用下面的方法归一化（但这不是必须的）：

```
// 设置引用的直方图
void setHistogram(const cv::Mat& h) {

   histogram= h;
   cv::normalize(histogram,histogram,1.0);
}
```

要反向投影直方图，只需指定图像、范围（这里我们假定所有通道的范围是相同的）和所用通道的列表。参见以下代码：

```
// 使用全部通道，范围[0,256]
cv::Mat find(const cv::Mat& image) {

  cv::Mat result;

  hranges[0]= 0.0;  // 默认范围[0,256]
  hranges[1]= 256.0;
  channels[0]= 0;   // 三个通道
```

```
  channels[1]= 1;
  channels[2]= 2;

  return find(image, hranges[0], hranges[1], channels);
}

// 查找属于直方图的像素
cv::Mat find(const cv::Mat& image,
              float minValue, float maxValue,
              int *channels) {

  cv::Mat result;

  hranges[0]= minValue;
  hranges[1]= maxValue;

  // 直方图的维度数与通道列表一致
  for (int i=0; i<histogram.dims; i++)
      this->channels[i]= channels[i];

  cv::calcBackProject(&image,
          1,          // 一次只使用一个图像
          channels,   // 向量表示哪个直方图维度属于哪个图像通道
          histogram,  // 用到的直方图
          result,     // 反向投影的图像
          ranges,     // 每个维度的值范围
          255.0       // 选用的换算系数
          // 把概率值从1映射到255
  );
}

  // 对反向投影结果做阈值化，得到二值图像
  if (threshold>0.0)
    cv::threshold(result, result,
              255.0*threshold, 255.0, cv::THRESH_BINARY);

  return result;
}
```

现在我们把前面用过的图像换成彩色版本（访问本书的网站查看彩色图像），并使用一个BGR直方图。这次我们要检测蓝天区域。首先我们装载彩色图像，定义兴趣区域，然后计算经过缩减的色彩空间上的3D直方图。代码如下：

```
// 装载彩色图像
ColorHistogram hc;
cv::Mat color= cv::imread("waves2.jpg");

// 提取兴趣区域
imageROI= color(cv::Rect(0,0,100,45)); // 蓝色天空区域

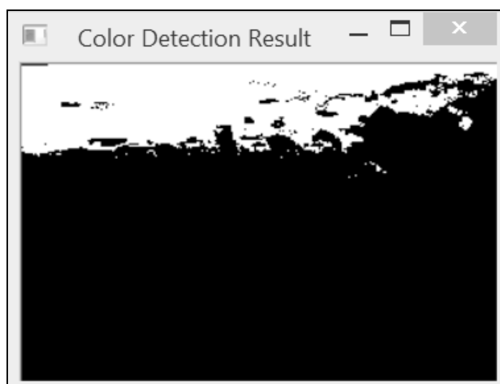
// 取得3D颜色直方图（每个通道含8个箱子）
hc.setSize(8); // 8 × 8 × 8
cv::Mat shist= hc.getHistogram(imageROI);
```

下一步是计算直方图，并用`find`方法检测图像中的天空区域：

```
// 创建内容搜寻器
ContentFinder finder;
// 设置用来反向投影的直方图
finder.setHistogram(shist);
finder.setThreshold(0.05f);

// 取得颜色直方图的反向投影
Cv::Mat result= finder.find(color);
```

上一节的彩色图像的检测结果如下：



通常来说，采用BGR色彩空间识别图像中的物体并不是最好的方法。这里为了提高可靠性，我们在计算直方图之前缩减了颜色的数量（要知道原始BGR色彩空间有超过1600万种的颜色）。提取的直方图代表了天空区域的典型颜色分布情况。用它在其他图像上反向投影，也能检测到天空区域。注意，用多个天空图像构建直方图，可以提高检测的准确性。

本例中，计算稀疏直方图可以减少内存使用量。你可以使用`cv::SparseMat`重做该实验。另外，如果要寻找色彩鲜艳的物体，使用HSV色彩空间的色调通道可能会更加有效。在其他情况下，最好使用感知上均匀的色彩空间（例如L*a*b*）的色度组件。

4.5.4　参阅

- 下一节我们将用HSV色彩空间检测图像中的物体。检测图像内容的方法很多，这是其中的一种。

4.6　均值平移算法查找目标

直方图反向投影的结果是一个概率分布图，表示一个指定图像片段出现在特定位置的概率。

假设我们已经知道图像中某个物体的大致位置，就可以用概率分布图找到物体的准确位置。最可能出现的位置就是窗口中概率最大的位置。因此我们可以从一个初始位置开始，在周围反复移动，就可能找到物体所在的准确位置。这个实现方法称为均值平移算法。

4.6.1 如何实现

假设我们已经是识别出一个感兴趣的物体——这里是狒狒的脸部——如下图所示（另见彩图6）：



这次我们采用HSV色彩空间的色调通道来描述物体。这意味着我们需要把图像转换成HSV色彩空间并提取色调通道，然后计算指定ROI的一维色调直方图。参见以下代码：

```
// 读取参考图像
cv::Mat image= cv::imread("baboon1.jpg");
// 狒狒脸部的ROI
cv::Mat imageROI= image(cv::Rect(110,260,35,40));
// 得到色调直方图
int minSat=65;
ColorHistogram hc;
cv::Mat colorhist=
         hc.getHueHistogram(imageROI,minSat);
```

我们在`ColorHistogram`类中增加了一个简便的方法，来获得色调直方图，代码如下：

```
// 计算一维色调直方图（带掩码）
// BGR的原图转换成HSV
// 忽略低饱和度的像素
cv::Mat getHueHistogram(const cv::Mat &image,
                        int minSaturation=0) {

  cv::Mat hist;

  // 转换成HSV色彩空间
```

```
  cv::Mat hsv;
  cv::cvtColor(image, hsv, CV_BGR2HSV);

  // 掩码（可用或可不用）
  cv::Mat mask;

  if (minSaturation>0) {

    // 把3个通道分割进3个图像
    std::vector<cv::Mat> v;
    cv::split(hsv,v);

    // 屏蔽低饱和度的像素
    cv::threshold(v[1],mask,minSaturation,255,
                               cv::THRESH_BINARY);
  }

  // 准备一维色调直方图的参数
  hranges[0]= 0.0;    // 范围为0~180
  hranges[1]= 180.0;
  channels[0]= 0;     // 色调通道

  // 计算直方图
  cv::calcHist(&hsv,
    1,         // 只有一个图像的直方图
    channels,  // 用到的通道
    mask,      // 二值掩码
    hist,      // 生成的直方图
    1,         // 这是一维直方图
    histSize,  // 箱子数量
    ranges     // 像素值范围
  );

  return hist;
}
```

然后把得到的直方图传给ContentFinder类的实例，如下：

```
ContentFinder finder;
finder.setHistogram(colorhist);
```

现在打开第二个图像，在它上面定位狒狒的脸部。首先需要把这个图像转换成HSV色彩空间，然后对第一个图像的直方图做反向投影。参见下面的代码：

```
image= cv::imread("baboon3.jpg");
// 转换成HSV色彩空间
cv::cvtColor(image, hsv, CV_BGR2HSV);
// 得到色调直方图的反向投影
int ch[1]={0};
finder.setThreshold(-1.0f); // 不做阈值化
cv::Mat result= finder.find(hsv,0.0f,180.0f,ch);
```

rect对象是一个初始矩形区域（即初始图像中狒狒脸部的位置），现在OpenCV的

cv::meanShift算法将会把它修改成狒狒脸部的新位置。参考以下代码：

```
// 窗口初始位置
cv::Rect rect(110,260,35,40);

// 用均值偏移法搜索物体
cv::TermCriteria criteria(cv::TermCriteria::MAX_ITER,
                          10,0.01);
cv::meanShift(result,rect,criteria);
```

脸部的初始位置（红色）和新位置（绿色）显示如下（另见彩图7）：



4.6.2 实现原理

本例中，为了突出被寻找物体的特征，我们使用了HSV色彩空间的色调分量。之所以这样做是因为狒狒脸部有非常独特的粉红色，使用像素的色调很容易标识狒狒脸部。因此第一步就是把图像转换成HSV色彩空间。使用CV_BGR2HSV标志转换图像后，得到的第一个通道就是色调分量。这是一个8位分量，值范围为0~180（如果使用cv::cvtColor，转换后的图像与原始图像的类型相同）。为了提取到色调图像，cv::split函数把三通道的HSV图像分割成三个单通道图像。这三个图像存放在一个std::vector实例中，并且色调图像是向量的第一个入口（即索引为0）。

在使用颜色的色调分量时，要把它的饱和度考虑在内（饱和度是矩向量的第二个入口），这一点通常很重要。如果颜色的饱和度很低，它的色调信息就会变得不稳定且不可靠。这是因为低饱和度颜色的B、G和R分量几乎是相等的。这导致很难确定它所表示的准确颜色。因此我们决定忽略低饱和度颜色的色彩分量。也就是不把它们统计进直方图中（在getHueHistogram方法中使用minSat参数，可屏蔽掉饱和度低于此阈值的像素）。

均值偏移算法是一个迭代过程，用于定位概率函数的局部最大值。定位的方法是寻找预定义窗口内部数据点的重心或加权平均值。然后把窗口移动到重心的位置，并重复该过程，直到窗口中心收敛到一个稳定的点。OpenCV实现该算法时定义了两个停止条件：迭代次数达到最大值；窗口中心的偏移值小于某个限值，可认为该位置收敛到一个稳定点。这两个条件存储在一个`cv::TermCriteria`实例中。`cv::meanShift`函数返回已经执行的迭代次数。显然，结果的好坏取决于在指定初始位置处提供的概率分布图的质量。注意，这里我们用颜色直方图表示图像的外观。也可以用其他特征的直方图（例如边界方向直方图）来表示物体。

4.6.3　参阅

- 均值偏移算法广泛应用于视觉追踪。第11章会详细探讨目标跟踪的问题。
- D. Comaniciu和P. Meer发表在《IEEE模式分析与机器智能》（*IEEE Transactions on Pattern Analysis and Machine Intelligence*）杂志2002年第5期第24卷上的文章“Mean Shift: A robust approach toward feature space analysis”首次提出了均值偏移算法。
- OpenCV也提供了CamShift算法的具体实现方法。这个算法是均值偏移算法的改进版本，它允许修改窗口的尺寸和方向。

4.7　比较直方图搜索相似图像

基于内容的图像检索是计算机视觉的一个重要课题。它包括根据一个已有的基准图像，找出一批内容相似的图像。我们知道，直方图是标识图像内容的一种有效方式，因此值得研究一下是否能用它来解决基于内容检索的问题。

这里的关键是要做到，仅仅比较它们的直方图就能测量出两个图像的相似度。需要定义一个测量函数来评估两个直方图之间的差异程度或相似程度。人们已经提出了很多这样的测量方法，OpenCV在`cv::compareHist`函数的实现过程中使用了其中的一些方法。

4.7.1　如何实现

为了将一个基准图像与一批图像进行对比并找出其中与它最相似的图像，我们创建了类`ImageComparator`。这个类引用了一个基准图像和一个输入图像（连同它们的直方图）。另外，因为我们要用颜色直方图来进行比较，因此用到了`ColorHistogram`类：

```
class ImageComparator {

  private:

    cv::Mat refH;          // 基准直方图
    cv::Mat inputH;        // 输入图像的直方图
```

```
    ColorHistogram hist; // 生成直方图
    int nBins; // 每个颜色通道使用的箱子数量

  public:

    ImageComparator() :nBins(8) {
}
```

为了得到更加可靠的相似度测量结果，我们需要在计算直方图时减少箱子的数量。可以在类中指定每个BGR通道所用的箱子数量。参见下面的代码：

```
// 设置比较直方图时使用的箱子数量
void setNumberOfBins( int bins) {

  nBins= bins;
}
```


用一个适当的设值函数指定基准图像，同时计算参考直方图。如下所示：

```
// 计算基准图像的直方图
void setReferenceImage(const cv::Mat& image) {

  hist.setSize(nBins);
  refH= hist.getHistogram(image);
}
```

最后，compare方法会将基准图像和指定的输入图像进行对比。下面的方法返回一个分数，表示两个图像的相似程度：

```
// 用BGR直方图比较图像
double compare(const cv::Mat& image) {

  inputH= hist.getHistogram(image);

  return cv::compareHist(refH,inputH,CV_COMP_INTERSECT);
}
```

前面的类可用来检索与给定的基准图像类似的图像。初始化类的实例时使用下面的代码：

```
ImageComparator c;
c.setReferenceImage(image);
```

这里我们用4.5节中海滩图的彩色版本作为基准图像，并将这个图像与后面的一系列图像进行对比。图像的显示顺序为相似度大的放前面，相似度小的放后面，如下所示：



4.7.2　实现原理

大多数直方图比较方法都是基于逐个箱子进行比较。正因为如此，在测量两个颜色直方图的相似度时减少直方图箱子数量显得十分重要。对`cv::compareHist`的调用非常简单。只需要输入两个直方图，函数就返回它们的差距。你可以通过一个标志参数指定想要使用的测量方法。`ImageComparator`类使用了交叉点方法（带有`CV_COMP_INTERSECT`标志）。该方法只是逐个箱子地比较每个直方图中的数值，并保存最小的值。然后把这些最小值累加，作为相似度测量值。因此，两个没有相同颜色的直方图得到交叉值为`0`，而两个完全相同的直方图得到的值就等于像素总数。

其他可用的算法有：卡方测量法（`CV_COMP_CHISQR`标志），累加各箱子的归一化平方差；关联性算法（`CV_COMP_CORREL`标志），基于信号处理中的归一化交叉关联操作符测量两个信号的相似度；Bhattacharyya测量法（`CV_COMP_BHATTACHARYYA`标志），用在统计学中，评估两个概率分布的相似度。

4.7.3　参阅

- OpenCV文档详细描述了不同的直方图比较方法中使用的公式。
- 推土机距离（Earth Mover Distance）是另一种流行的直方图比较方法，在OpenCV中通过`cv::EMD`函数实现。这种方法的主要优势，是它在评估两个直方图的相似度时，考虑了在邻近箱子中发现的数值。具体描述可查看Y. Rubner、C. Tomasi和L. J. Guibas发表在*Int. Journal of Computer Vision* 2000年第2期卷40（页码99~121）上的“The Earth Mover's Distance as a Metric for Image Retrieval”。

4.8 用积分图像统计像素

前面几节讲了直方图的计算方法，即遍历图像的全部像素并累计每个强度值在图像中出现的次数。我们也看到，有时只需要计算图像中某个特定区域的直方图。实际上累计图像的某个子区域内的像素总数，是很多计算机视觉算法中常见的过程。现在假设需要对图像中的多个兴趣区域计算几个此类直方图。这些计算过程都马上会变得非常耗时。这种情况下，有一个工具可以极大地提高统计图像子区域像素的效率：积分图像。

在统计图像兴趣区域的像素时，使用积分图像是一种高效的方法。它在程序中的应用非常广泛，例如用于计算基于不同大小的滑动窗口。

本节将讲解积分图像背后的原理。这里的目标是说明如何只用三次算术运算，累加一个矩形区域的像素。在我们学会这个概念后，4.8.3节将展示两个有效使用积分图像的实例。

4.8.1 如何实现

本节将使用下面的图像来做演示，识别出图像中的一个兴趣区域。区域内容为一个骑自行车的女孩：



在累加多个图像区域的像素时，积分图像显得非常有用。通常来说，要获得兴趣区域全部像素的累加和，常规的代码如下：

```
// 打开图像
cv::Mat image= cv::imread("bike55.bmp",0);
// 定义图像的ROI（这里为骑自行车的女孩）
int xo=97, yo=112;
int width=25, height=30;
cv::Mat roi(image,cv::Rect(xo,yo,width,height));
```

```
// 计算累加值
// 返回一个多通道图像下的Scalar数值
cv::Scalar sum= cv::sum(roi);
```

cv::sum函数只是遍历区域内的所有像素，并计算累加和。使用积分图像后，只需要三次加法运算即可实现该功能。不过首先需要计算积分图像，代码如下：

```
// 计算积分图像
cv::Mat integralImage;
cv::integral(image,integralImage,CV_32S);
```

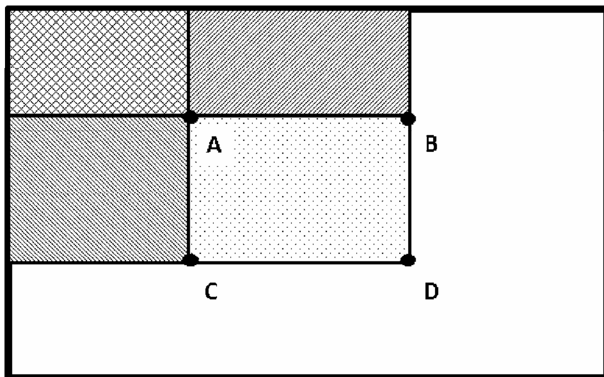
可以在积分图像上用简单的算术表达式获得同样的结果（下一节会详细解释），代码为：

```
// 用三个加/减运算得到一个区域的累加值
int sumInt= integralImage.at<int>(yo+height,xo+width)
          -integralImage.at<int>(yo+height,xo)
          -integralImage.at<int>(yo,xo+width)
          +integralImage.at<int>(yo,xo);
```

两种做法得到的结果是一样的。但计算积分图像需要遍历全部像素，因此速度比较慢。关键是一旦这个初始计算完成，只需要添加四个像素就能得到兴趣区域的累加和，与区域的尺寸无关。因此，如果需要在多个尺寸的区域上计算像素累加和，就最好采用积分图像。

4.8.2　实现原理

上一节我们简单地演示了积分图像的“神奇”功能，即可用来快速计算矩形区域内的像素累加和，并通过演示简要介绍了积分图像的概念。为了理解积分图像的实现原理，我们先对它下一个定义。取图像左上侧的全部像素计算累加和，并用这个累加和替换图像中的每一个像素，用这种方式得到的图像称为积分图像。计算积分图像时只需对图像扫描一次。这是因为当前像素的积分值等于上一像素的积分值加上当前行的累计值。因此积分图像就是一个包含像素累加和的新图像。为防止溢出，积分图像的值通常采用int类型（CV_32S）或float类型（CV_32F）。例如下图中，积分图像的像素A包含左上角区域，即双阴影线图案标识的区域的像素的累加和。



计算完积分图像后，只需要访问四个像素就可以得到任何矩形区域的像素累加和。这里解释一下原因。再来看前面的图片，计算由A、B、C、D四个像素表示区域的像素累加和，先读取D的积分值，然后减去B的像素值和C的左手边区域的像素值。但是这样就把A左上角的像素累加和减了两次，因此需要重新加上A的积分值。所以计算A、B、C、D区域内的像素累加的正式公式为：A − B − C + D。如果用cv::Mat方法访问像素值，公式可转换成以下代码：

```
//窗口的位置是(xo,yo)，尺寸是width × height
return (integralImage.at<cv::Vec<T,N>>
                              (yo+height,xo+width)
      -integralImage.at<cv::Vec<T,N>>(yo+height,xo)
      -integralImage.at<cv::Vec<T,N>>(yo,xo+width)
      +integralImage.at<cv::Vec<T,N>>(yo,xo));
```

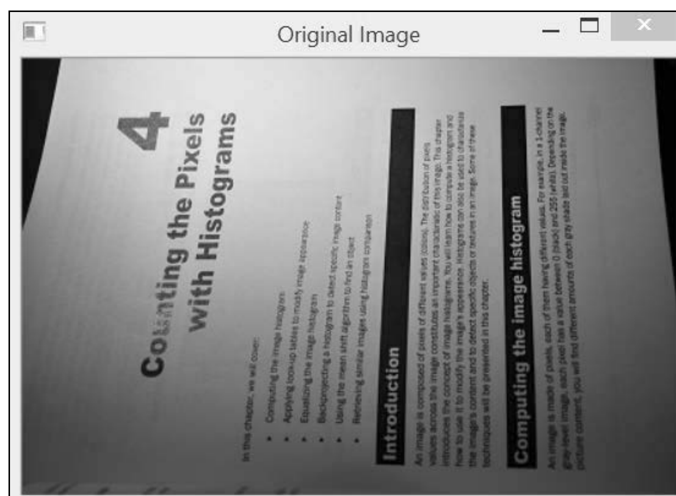
不管兴趣区域的尺寸有多大，使用这种方法计算的复杂度是恒定不变的。注意，为了简化，这里使用了cv::Mat类的at方法，它访问像素值的效率并不是最高的（参见第2章）。4.8.3节将讨论这方面内容，通过两个例子说明积分图像在效率上的优势。

4.8.3　扩展阅读

积分图像适合用来执行多次像素累计值的统计。本段将通过介绍自适应阈值化的概念，说明积分图像的使用方法。在需要快速计算多个窗口的直方图时，积分图像非常有用。本节也将对此进行解释。

1. 自适应的阈值化

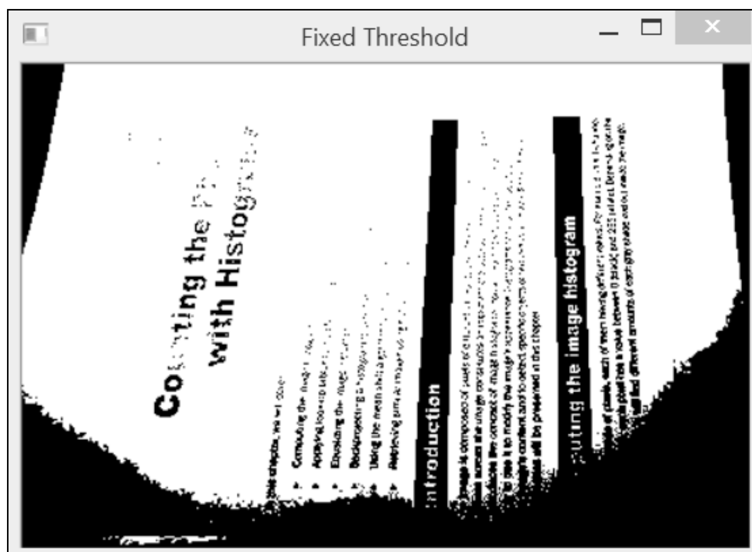
通过对图像应用阈值来创建二值图像是从图像中提取有意义元素的好方法。假设有下面这个关于本书的图像：



为了分析图像中的文字，我们对该图像应用一个阈值，代码如下：

```
// 使用固定的阈值
cv::Mat binaryFixed;
cv::threshold(image,binaryFixed,70,255,cv::THRESH_BINARY);
```

得到如下结果：



实际上，不管选用什么阈值，图像都会丢失一部分文本，还有部分文本会消失在阴影下。要解决这个问题，有一个办法就是采用局部阈值，即根据每个像素的邻域计算阈值。这种策略称为*自适应阈值化*，包括将每个像素的值与邻域的平均值进行比较。如果某像素的值与它的局部平均值差别很大，就会被当作异常值在阈值化过程中剔除。

因此自适应阈值化需要计算每个像素周围的局部平均值。这需要多次计算图像窗口的累计值，可以通过积分图像提高计算效率。正因为如此，方法的第一步就是计算积分图像：

```
// 计算积分图像
cv::Mat iimage;
cv::integral(image,iimage,CV_32S);
```

现在我们可以遍历全部像素，并计算方形邻域的平均值。我们也可以使用`IntegralImage`类来实现这个功能，但是这个类在访问像素时使用了效率很低的`at`方法。根据第2章学过的方法，我们可以使用指针遍历图像以提高效率。循环代码如下：

```
int blockSize= 21; // 邻域的尺寸
int threshold=10;  // 像素将与(mean-threshold)进行比较

// 逐行
```

```
int halfSize= blockSize/2;
  for (int j=halfSize; j<nl-halfSize-1; j++) {

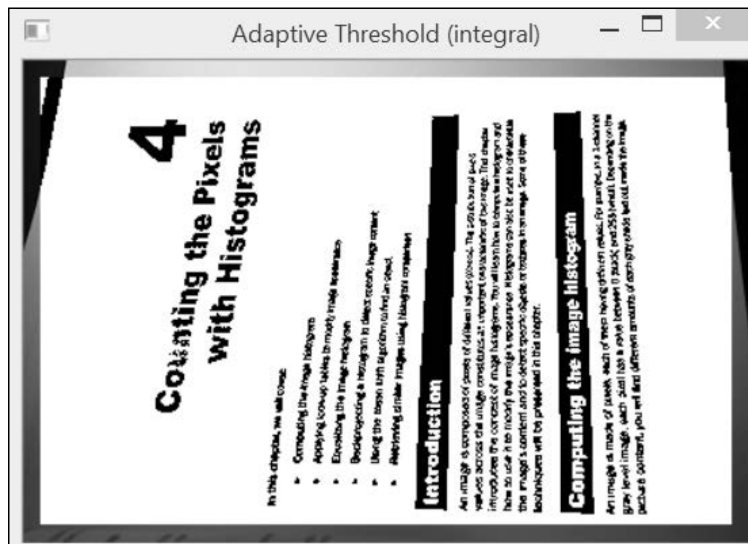
    // 得到第j行的地址
    uchar* data= binary.ptr<uchar>(j);
    int* idata1= iimage.ptr<int>(j-halfSize);
    int* idata2= iimage.ptr<int>(j+halfSize+1);

    // 一个线条的像素
    for (int i=halfSize; i<nc-halfSize-1; i++) {

      // 计算累加值
      int sum= (idata2[i+halfSize+1]-
                   idata2[i-halfSize]-
               idata1[i+halfSize+1]+
                   idata1[i-halfSize])/
                      (blockSize*blockSize);

      // 应用自适应阈值
      if (data[i]<(sum-threshold))
        data[i]= 0;
      else
        data[i]=255;
    }
  }
```


本例使用了21 × 21的邻域。为计算每个平均值，我们需要访问界定正方形邻域的四个积分像素：两个在标有`idata1`的线条上，另两个在标有`idata2`的线条上。当前像素与计算得到的平均值进行比较。为了确保被剔除的像素与局部平均值有明显的差距，这个平均值要减去阈值（这里设为`10`）。由此得到下面的二值图像：



很明显，这比我们用固定阈值得到的结果好得多。自适应阈值化是一种常用的图像处理技术。OpenCV中也实现了这种方法：

```
cv::adaptiveThreshold(image,          // 输入图像
        binaryAdaptive,               // 输出二值图像
        255,                          // 输出的最大值
        cv::ADAPTIVE_THRESH_MEAN_C,   // 方法
        cv::THRESH_BINARY,            // 阈值类型
        blockSize,                    // 块的大小
        threshold);                   // 使用的阈值
```

调用这个函数得到的结果与使用积分图像的结果完全相同。另外，除了在阈值化中使用局部平均值，本例中的函数还可以使用高斯（Gaussian）加权累计值（该方法的标志为ADAPTIVE_THRESH_GAUSSIAN_C）。有意思的是，这种实现方式要比调用cv::adaptiveThreshold稍微快一些。

最后需要注意，我们也可以用OpenCV的图像运算符来编写自适应阈值化过程。具体方法如下：

```
cv::Mat filtered;
cv::Mat binaryFiltered;
cv::boxFilter(image,filtered,CV_8U,
              cv::Size(blockSize,blockSize));
filtered= filtered-threshold;
binaryFiltered= image>= filtered;
```

图像滤波的内容将在第6章介绍。

2. 用直方图实现视觉追踪

通过前面几节的学习，我们知道可用直方图表示物体外观的全局特征。本节我们搜寻一个所呈现直方图与目标物体相似的图像区域，演示如何在图像中定位物体，以此说明积分图像的用途。我们在4.6节实现了这个功能，用的是直方图反向投影概念和通过均值偏移局部搜索的方法。这次我们在整幅图像上显式地搜索具有类似直方图的区域，以此找到物体。

有一种特殊情况，即由0和1组成的二值图像生成积分图像，这时的积分累计值就是指定区域内值为1的像素总数。本节将利用这一现象计算灰度图像的直方图。

cv::integral函数也可用于多通道图像。可以充分利用这点，用积分图像计算图像子区域的直方图。只需简单地把图像转换成由二值平面组成的多通道图像。每个平面关联直方图的一个箱子，并显示哪些像素的值会进入该箱子。下面的函数从一个灰度图像创建这样的多图层图像：

```
// 转换成二值图层组成的多通道图像
// nPlanes必须是2的幂
void convertToBinaryPlanes(const cv::Mat& input,
                           cv::Mat& output, int nPlanes) {

    // 需要屏蔽的位数
```

```
    int n= 8-static_cast<int>(
      log(static_cast<double>(nPlanes))/log(2.0));
    // 用来消除最低有效位的掩码
    uchar mask= 0xFF<<n;

    // 创建二值图像的向量
    std::vector<cv::Mat> planes;
    // 消除最低有效位，箱子数减为nBins
    cv::Mat reduced= input&mask;

    // 计算每个二值图像平面
    for (int i=0; i<nPlanes; i++) {

      // 将每个等于i<<shift的像素设为1
      planes.push_back((reduced==(i<<n))&0x1);
    }

    // 创建多通道图像
    cv::merge(planes,output);
}
```


你也可以把积分图像的计算过程封装进模板类中：

```
template <typename T, int N>
class IntegralImage {

    cv::Mat integralImage;

  public:

    IntegralImage(cv::Mat image) {

    // （很耗时）计算积分图像
    cv::integral(image,integralImage,cv::DataType<T>::type);
    }

    // 通过访问4个像素，计算任何尺寸子区域的累计值
    cv::Vec<T,N> operator()(int xo, int yo,
                             int width, int height) {

    // （xo, yo）处的窗口，尺寸为width × height
    return (integralImage.at<cv::Vec<T,N>>
                                   (yo+height,xo+width)
        -integralImage.at<cv::Vec<T,N>>(yo+height,xo)
        -integralImage.at<cv::Vec<T,N>>(yo,xo+width)
        +integralImage.at<cv::Vec<T,N>>(yo,xo));
    }
};
```

我们在前面的图像中识别了骑车女孩，现在要在后面的图像中找到她。首先计算原始图像中女孩的直方图，可通过本章前面创建的Histogram1D类实现。以下代码生成16个箱子的直方图：

```
// 16个箱子的直方图
Histogram1D h;
h.setNBins(16);
// 计算图像中ROI的直方图
cv::Mat refHistogram= h.getHistogram(roi);
```

这个直方图将作为基准，在下面的图像中定位目标（即骑车女孩）。

假设我们仅有的信息，是图像中女孩在水平方向移动。因为需要对不同的位置计算很多直方图，我们先做准备工作，即计算积分图像。参见以下代码：

```
// 首先创建16个平面的二值图像
cv::Mat planes;
convertToBinaryPlanes(secondIimage,planes,16);
// 然后计算积分图像
IntegralImage<float,16> intHistogram(planes);
```

执行搜索时，循环遍历可能出现目标的位置，并将它的直方图与基准直方图作比较，目的是找到与直方图最相似的位置。参见以下代码：

```
double maxSimilarity=0.0;
int xbest, ybest;
// 遍历原始图像中女孩位置周围的水平长条
for (int y=110; y<120; y++) {
  for (int x=0; x<secondImage.cols-width; x++) {

    // 用积分图像计算16个箱子的直方图
    histogram= intHistogram(x,y,width,height);
    // 计算与基准直方图的差距
    double distance= cv::compareHist(refHistogram,
                          histogram, CV_COMP_INTERSECT);
    // 找到最相似直方图的位置
    if (distance>maxSimilarity) {

      xbest= x;
      ybest= y;
      maxSimilarity= distance;
    }
  }
}
// 在最准确的位置画矩形
cv::rectangle(secondImage,
              cv::Rect(xbest,ybest,width,height),0));
```

然后就可确定直方图最相似的位置，如下图所示：



白色矩形表示搜索的区域。计算区域内部所有窗口的直方图。这里窗口尺寸是固定的，但是更好的做法是也搜索稍小或稍大的窗口，以便应对缩放比例可能出现的变动。需要注意，为了减少计算复杂度，要减少直方图箱子的计算数量。本例中减少到16个箱子。因此在这个多平面图像中，平面0包含一个二值图像，表示值从0到15的所有像素；平面1表示值从16到31的全部像素，等等。

对物体的搜索过程包含了用预定范围的像素计算指定尺寸的所有窗口的直方图的计算过程。这意味着从积分图像对3200个不同直方图进行了高效计算。`IntegralImage`类返回的直方图都存储在`cv::Vec`对象中（因为用了`at`方法）。然后用`cv::compareHist`函数找到最相似的直方图（和大多数OpenCV函数一样，这个函数可以利用实用的通用参数类型`cv::InputArray`获得`cv::Mat`或`cv::Vec`）。

4.8.4　参阅

- 第8章将讲述`SURF`运算符，它也基于对积分图像的使用。
- A. Adam、E. Rivlin和I. Shimshoni发表在*proceeding of the Int. Conference on Computer Vision and Pattern Recognition* 2006年第798-805页的文章“Robust Fragments-based Tracking using the Integral Histogram”介绍了一种有趣的方法，即利用积分图像在一个图像队列中跟踪物体。

4

第 5 章

用形态学运算变换图像

本章包括以下内容：

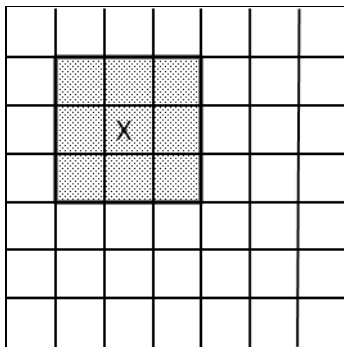
- 用形态学滤波器腐蚀和膨胀图像；
- 用形态学滤波器开启和闭合图像；
- 用形态学滤波器检测边缘和角点；
- 用分水岭算法实现图像分割；
- 用MSER算法提取特征区域；
- 用GrabCut算法提取前景物体。

5.1 简介

数学形态学是一门20世纪60年代发展起来的理论，用于分析和处理离散图像。它定义了一系列运算，用预先定义的形状元素探测图像，从而实现图像的转换。这个结构元素与像素领域的相交方式决定了运算的结果。本章介绍几种最重要的形态学运算，并探讨用基于形态学运算的算法进行图像分割和特征检测的问题。

5.2 形态学滤波器腐蚀和膨胀图像

腐蚀和膨胀是最基本的形态学运算，因此把它们放在第一节介绍。数学形态学中最基本的概念是*结构元素*。结构元素可以简单地定义为像素的组合（下图的正方形），在对应的像素上定义了一个原点（也称*锚点*）。形态学滤波器的应用过程就包含了用这个结构元素探测图像中每个像素的操作过程。把某个像素设为结构元素的原点后，结构元素和图像重叠部分的像素集（下图的九个阴影像素）就是特定形态学运算的应用对象。结构元素原则上可以是任何形状，但通常它是一个简单形状，如正方形、圆形或菱形，并且把中心点作为原点（主要是因为效率），如下图：



5.2.1　准备工作

因为形态学滤波器通常作用于二值图像，所以我们采用4.1节中通过阈值化创建的二值图像。但在形态学中，我们习惯用高像素值（白色）表示前景物体，用低像素值（黑色）表示背景物体，因此我们对图像做了反向处理。

在形态学术语中，下面的图像称为第4章所建图像的补码：



5.2.2　如何实现

OpenCV用简单的函数实现腐蚀和膨胀运算，即`cv:erode`和`cv:dilate`。它们的用法很简单：

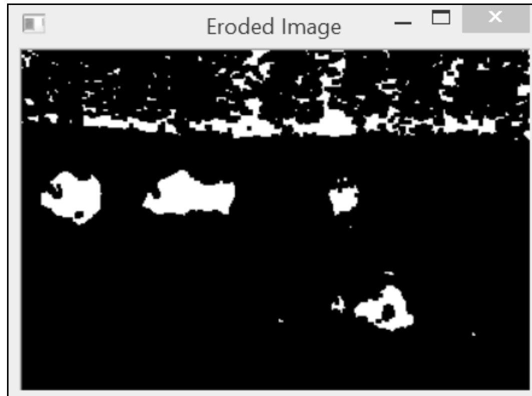
```
// 读取输入图像
cv::Mat image= cv::imread("binary.bmp");

// 腐蚀图像
cv::Mat eroded; // 目标图像
```

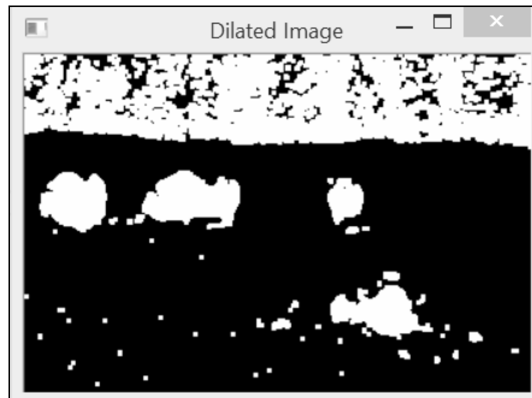
```
cv::erode(image,eroded,cv::Mat());

// 膨胀图像
cv::Mat dilated; // 目标图像
cv::dilate(image,dilated,cv::Mat());
```

这些函数生成的两个图像如下所示。第一个截图是腐蚀图像：



第二个截图是膨胀图像：



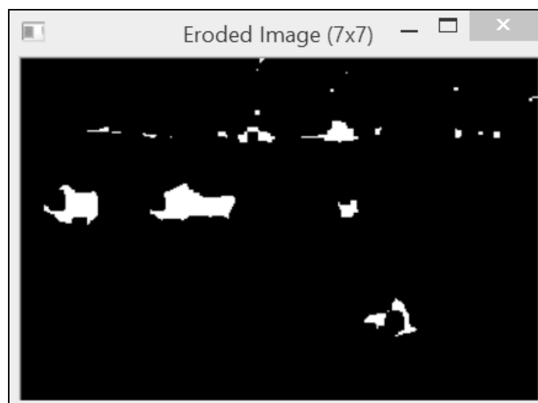
5.2.3　实现原理

和其他形态学滤波器一样，本节的两个滤波器在每个像素周围的像素集（或相邻像素）上操作，具体由结构元素定义。在某个像素上应用结构元素时，结构元素的锚点与该像素对齐，所有与结构元素相交的像素就包含在当前集合中。腐蚀就是把当前像素替换成所定义像素集合中的最小像素值。膨胀是腐蚀的反运算，它把当前像素替换成所定义像素集合中的最大像素值。由于输入的二值图像只包含黑色（`0`）和白色（`255`）像素，因此每个像素都会被替换成白色或黑色像素。

要形象化地理解这两种运算的作用，可考虑背景（黑色）和前景（白色）的物体。腐蚀时，如果结构元素放到某个像素位置时碰到了背景（即交集中有一个像素是黑色的），那么这个像素就变为背景。膨胀时，如果结构元素放到某个背景像素位置时碰到了前景物体，那么这个像素就标为白色。正因为如此，腐蚀后的图像中物体尺寸会缩小（形状被腐蚀）。注意有些面积较小的物体（可看作是背景中的“噪声”像素）会彻底消失。类似地，膨胀后的物体会变大，而物体中有些“空隙”会被填满。OpenCV默认使用3 × 3正方形结构元素。在调用函数时，参考前面的例子将第三个参数指定为空矩阵（即`cv::Mat()`），就能得到默认的结构元素。你也可以通过提供一个矩阵来指定结构元素的大小（以及形状），矩阵中的非零元素即构成结构元素。下面的例子使用7 × 7的结构元素：

```
cv::Mat element(7,7,CV_8U,cv::Scalar(1));
cv::erode(image,eroded,element);
```

这次的结果更有破坏性，如下图所示：



还有一种方法能得到同样的结果，就是在图像上反复地应用同一个结构元素。这两个函数都有一个用于指定重复次数的可选参数：

```
// 腐蚀图像三次
cv::erode(image,eroded,cv::Mat(),cv::Point(-1,-1),3);
```

参数`cv::Point(-1,-1)`表示原点是矩阵的中心点（默认值），可以定义结构元素上的任何位置。由此得到的图像与使用7 × 7结构元素得到的图像是一样的。实际上，对图像腐蚀两次相当于对结构元素自身膨胀后的图像进行腐蚀。这个规则也适用于膨胀。

最后，鉴于前景/背景概念有很大的随意性，我们可得到以下的实验结论（这是腐蚀/膨胀运算的基本性质）。用结构元素腐蚀前景物体可看作对图像背景部分的膨胀，因此我们可以得出如下实验结论：

- 腐蚀图像相当于对其反色图像膨胀后再取反色；

- 膨胀图像相当于对其反色图像腐蚀后再取反色。

5.2.4　扩展阅读

虽然这里我们将形态学滤波器应用在二值图像上，但这些滤波器也能应用在灰度图像，甚至彩色图像上，并且方法的定义是相同的。

另外，OpenCV的形态学函数支持就地处理。这意味着输入图像和输出图像可以采用同一个变量，如下：

```
cv::erode(image,image,cv::Mat());
```

OpenCV会创建必需的临时图像，保证这种方法能正常运行。

5.2.5　参阅

- 5.3节按顺序使用腐蚀和膨胀滤波器，产生新的运算符；
- 5.4节在灰度图像上应用形态学滤波器。

5.3　用形态学滤波器开启和闭合图像

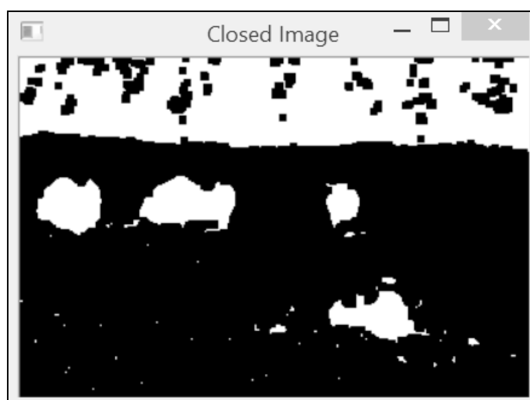
上一节介绍了两种基本的形态学运算：腐蚀和膨胀。我们可以利用它们定义新的运算。接下来两节将讲解其中的几种运算。本节讲解开启和闭合运算。

5.3.1　如何实现

为了应用较高级别的形态学滤波器，需要用`cv::morphologyEx`函数，并传入对应的函数代码。例如下面的调用方法将适用于闭合运算：

```
cv::Mat element5(5,5,CV_8U,cv::Scalar(1));
cv::Mat closed;
cv::morphologyEx(image,closed,cv::MORPH_CLOSE,element5);
```

注意，为了让滤波器的效果更加明显，这里我们使用了5×5的结构元素。如果输入上节的二值图像，得到的图像类似于下图：



类似地，应用形态学开启运算后将得到如下图像：



得到上面图像的代码是：

```
cv::Mat opened;
cv::morphologyEx(image,opened,cv::MORPH_OPEN,element5);
```

5.3.2 实现原理

开启和闭合滤波器的定义，只是简单地使用了基本的腐蚀和膨胀运算。闭合的定义是对图像先膨胀后腐蚀。开启的定义是对图像先腐蚀后膨胀。

因此可以用以下方法对图像做闭合运算：

```
// 膨胀原图像
cv::dilate(image,result,cv::Mat());
// 就地腐蚀膨胀后的图像
cv::erode(result,result,cv::Mat());
```

调换这两个函数的调用次序，就得到开启滤波器。查看闭合滤波器的结果，可看到白色的前

景物体中的小空隙已经被填满。闭合滤波器也会把邻近的物体连接起来。基本上，所有小到不能完整容纳结构元素的空隙或间隙，都会被闭合滤波器消除。

与闭合相反，开启滤波器消除了背景中的几个小物体。所有小到不能容纳结构元素的物体都会被移除。

这些滤波器常用于目标检测。闭合滤波器可把错误分裂成小碎片的物体连接起来，而开启滤波器可以移除因图像噪声产生的斑点。因此最好在使用这些滤波器时，按一定的顺序调用。在把我们测试用的二值图像成功地闭合和开启后，得到的图像只显示场景中的主要物体，如下图所示。如果优先考虑过滤噪音，可以先开启后闭合，但这样做的坏处是会消除掉部分物体碎片。



注意，对一个图像进行多次同样的开启运算（闭合运算也类似）是没有作用的。事实上，因为第一次使用开启滤波器时已经填充了空隙，再使用同一个滤波器将不会使图像产生变化。用数学术语讲，这些运算是幂等（idempotent）的。

5.3.3　参阅

在提取图像中的连通组件前，通常要用开启和闭合运算来清理图像。7.5节会详细解释这点。

5.4　用形态学滤波器检测边缘和角点

形态学滤波器也可用于检测图像中的某些特征。本节我们将学习如何检测灰度图像的边缘和角点。

5.4.1　准备工作

本节使用这个图像：



5.4.2 如何实现

可以用cv::morphologyEx函数的相关滤波器，检测图像中的边缘。参见以下代码：

```
// 用3 × 3结构元素得到梯度图像
cv::Mat result;
cv::morphologyEx(image,result,
                         cv::MORPH_GRADIENT,cv::Mat());

// 对图像阈值化得到一个二值图像
int threshold= 40;
cv::threshold(result, result,
                    threshold, 255, cv::THRESH_BINARY);
```

得到的结果如下图所示：



为了用形态学检测角点，现在我们定义MorphoFeatures类：

```
class MorphoFeatures {

  private:

      // 用于产生二值图像的阈值
      int threshold;
      // 用于检测角点的结构元素
      cv::Mat_<uchar> cross;
      cv::Mat_<uchar> diamond;
      cv::Mat_<uchar> square;
      cv::Mat_<uchar> x;
```

使用形态学检测角点的过程比较复杂，需要连续使用多个不同的形态学滤波器。这是一个使用非正方形结构元素的典型案例。事实上，这需要在构造函数中定义四个不同的结构元素，分别是正方形、菱形、十字形和X形（为了简化，所有结构元素的尺寸都固定为5×5）：

```
MorphoFeatures() : threshold(-1),
      cross(5, 5), diamond(5, 5), square(5, 5), x(5, 5) {

      // 创建十字形结构元素
      cross <<
        0, 0, 1, 0, 0,
        0, 0, 1, 0, 0,
        1, 1, 1, 1, 1,
        0, 0, 1, 0, 0,
        0, 0, 1, 0, 0;
      // 用类似方法创建其他结构元素
```

在角点特征的检测过程中，要依次应用所有结构元素，得到角点分布图：

```
cv::Mat getCorners(const cv::Mat &image) {

   cv::Mat result;

   // 用十字形元素膨胀
   cv::dilate(image,result,cross);

   // 用菱形元素腐蚀
   cv::erode(result,result,diamond);

   cv::Mat result2;
   // 用X形元素膨胀
   cv::dilate(image,result2,x);

   // 用正方形元素腐蚀
   cv::erode(result2,result2,square);

   // 比较两个经过闭合运算的图像，得到角点
   cv::absdiff(result2,result,result);
```

```
    // 应用阈值，获得二值图像
    applyThreshold(result);

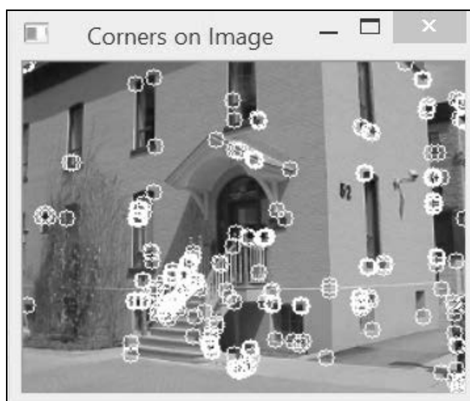
    return result;
}
```

然后可用以下代码检测图像中的角点：

```
// 得到角点
cv::Mat corners;
corners= morpho.getCorners(image);

// 在图像上显示角点
morpho.drawOnImage(corners,image);
cv::namedWindow("Corners on Image");
cv::imshow("Corners on Image",image);
```

图像中的角点以圆形标注，如下图所示：



5.4.3　实现原理

有一个方法可帮助理解灰度图像上形态学运算的效果，就是把图像看作是一个拓扑地貌，不同的灰度级别代表不同的高度（或海拔）。基于这种观点，明亮的区域代表高山，黑暗的区域代表深谷。边缘相当于黑暗和明亮像素之间的快速过渡，因此可以把边缘比喻成陡峭的悬崖。腐蚀这种地形的最终结果是：每个像素被替换成特定领域内的最小值，从而降低它的高度。结果是悬崖被“腐蚀”，山谷扩大。膨胀的效果刚好相反，即悬崖扩大，山谷缩小。但不管哪种情况，平地（即强度值固定的区域）都会保持相对不变。

根据这个结论，可以得到一种检测图像边缘（或悬崖）的简单方法，即通过计算膨胀后的图像与腐蚀后的图像之间的的差距得到边缘。因为这两种转换后图像的差别主要在边缘位置，它们相减后，边缘会很明显。在`cv::morphologyEx`函数中输入`cv::MORPH_GRADIENT`参数，即可实现此功能。显然，结构元素越大，检测到的边缘就越宽。这种边缘检测运算也叫Beucher梯度

（下一章将详细讨论图像梯度的概念）。注意还有两种简单的方法能得到类似结果，即膨胀后的图像减去原始图像，或者原始图像减去腐蚀后的图像。那样得到的边缘会更窄。

角点检测使用了四个不同的结构元素，因此检测过程更为复杂。OpenCV并没有实现这个运算，但我们把它放这里，是为了说明如何定义和组合不同形状的结构元素。它的原理是通过使用两个不同的结构元素膨胀和腐蚀图像，从而实现图像的闭合运算。选用这些结构元素后，直线的边缘保持不变。但因为它们各自的作用，拐角处的边缘会受影响。下面的图像很简单，由单个白色正方形构成，我们用它更好地理解不对称闭合运算的效果：



第一个正方形是原始图像。用十字形结构元素膨胀后，正方形的边缘扩大了，但拐角处没有扩大，因为在拐角处十字形不会形成正方形。上图中间的正方形就对应这种结果。然后用菱形的结构元素腐蚀膨胀后的图像。这个腐蚀运算把大部分边缘都推回到原始位置，但角点因未被膨胀会被推得更远。这时得到最右边的正方形，可以看到它已经失去了尖角。用X形和正方形结构元素重复上述过程。这两个元素与前面两个类似，只是旋转了角度，因此它们可捕获旋转了45度的角点。最后比较两个结果，就可提取到角点特征。

5.4.4　参阅

- 6.4节介绍了用于检测边缘的其他滤波器。
- 第8章展示了不同的角点检测算子。
- J.-F. Rivest、P. Soille和S. Beucher发表在*ISET's symposium on electronic imaging science and technology, SPIE* 1992年2月刊上的文章“The Morphological gradients”详细论述了形态学梯度的概念。
- F.Y. Shih、C.-F. Chuang和V. Gaddipati发表在*Pattern Recognition Letters* 2005年5月第26卷第7期上的文章“A modified regulated morphological corner detector”提供了更多有关于形态学角点检测的信息。

5.5　用分水岭算法实现图像分割

分水岭变换是一种流行的图像处理算法，用于快速将图像分割成多个同质区域。它基于这样的思想：如果把图像看作一个拓扑地貌，那么同类区域就相当于陡峭的边缘内相对平坦的盆地。

因为算法很简单，它的原始版本会过度分割图像，产生很多小的区域。因此OpenCV提出了该算法的改进版本，使用一系列预定义标记来引导图像分割的定义方式。

5.5.1 如何实现

使用分水岭分割法，需要调用cv::watershed函数。该函数的输入对象是一个标记图像，图像的像素值为32位有符号整数，每个非零像素代表一个标签。它的原理是对图像中部分像素作标记，表明它们的所属区域是已知的。分水岭算法可根据这个初始标签确定其他像素所属的区域。在本节我们将首先建立一个标记图像作为灰度图像，然后将其转换成整型图像。我们把这个步骤封装进WatershedSegmenter类。参见以下代码：

```
class WatershedSegmenter {

  private:

     cv::Mat markers;

  public:

     void setMarkers(const cv::Mat& markerImage) {

      // 转换成整数型图像
      markerImage.convertTo(markers,CV_32S);
     }

     cv::Mat process(const cv::Mat &image) {

      // 应用分水岭
      cv::watershed(image,markers);

      return markers;
     }
```

在不同的程序中获得标记的方式各不相同。例如，可在预处理过程中识别出一些属于某个兴趣物体的像素。然后可根据初始检测结果，使用分水岭算法划出整个物体的边缘。本节我们将利用本章一直使用的二值图像，识别出对应原始图像中的动物（原始图像见第4章开头部分）。因此我们需要从二值图像中识别出属于前景（动物）的像素以及属于背景（主要是草地）的像素。这里我们把前景像素标记为255，背景像素标记为128（该数字是随意选择的。任何不等于255的数字都可以使用）。其他像素的标签是未知的，标记为0。

现在，这个二值图像包含了太多属于图像不同部分的白色像素，因此要对图像做深度腐蚀运算，只保留属于重点物体的像素：

```
// 消除噪声和细小物体
cv::Mat fg;
cv::erode(binary,fg,cv::Mat(),cv::Point(-1,-1),4);
```

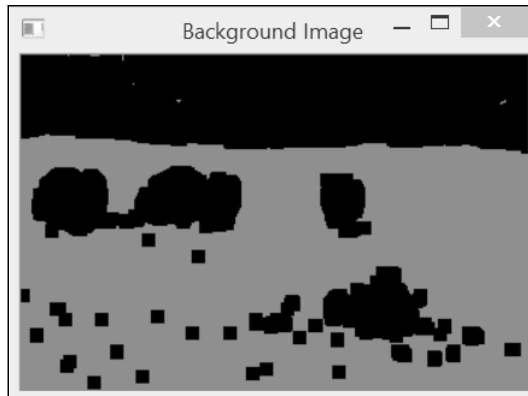
得到的图像如下：



注意，仍然有少量属于背景（森林）的像素保留了下来，不用管它们，可将它们看作兴趣物体。类似地，我们通过对原二值图像做一次大幅度的膨胀运算来选中一些背景像素：

```
// 标识不含物体的图像像素
cv::Mat bg;
cv::dilate(binary,bg,cv::Mat(),cv::Point(-1,-1),4);
cv::threshold(bg,bg,1,128,cv::THRESH_BINARY_INV);
```

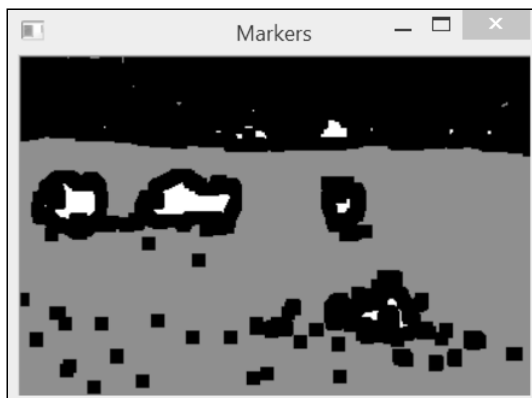
得到的黑色像素对应背景像素。因此在膨胀后，要立即通过阈值化运算把它们赋值为128。得到的图像如下：



合并这两个图像得到标记图像，代码为：

```
// 创建标记图像
cv::Mat markers(binary.size(),CV_8U,cv::Scalar(0));
markers= fg+bg;
```

注意这里是如何用重载运算符+来合并图像的。下面的图像将被输入分水岭算法：

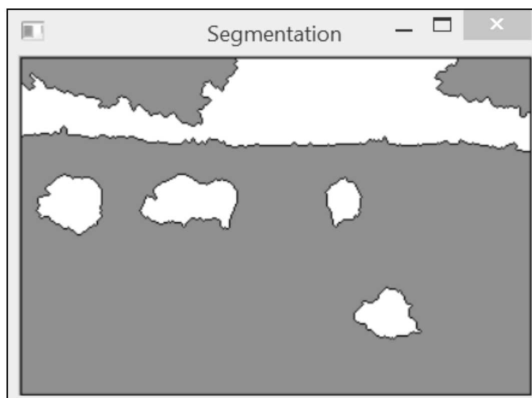


毫无疑问，在这个输入图像中，白色区域属于前景物体，灰色区域属于背景，而黑色区域带有未知标签。然后可用下面的方法来分割图像：

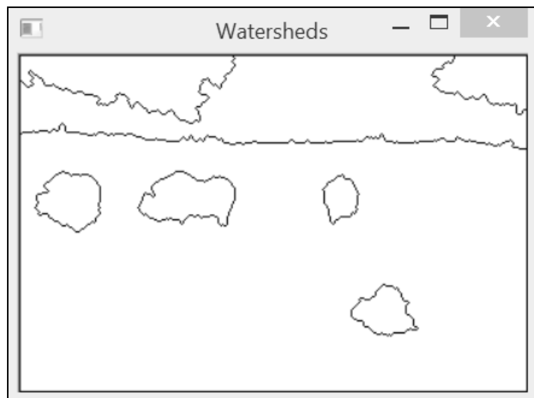
```
// 创建分水岭分割类的对象
WatershedSegmenter segmenter;

// 设置标记图像，然后执行分割过程
segmenter.setMarkers(markers);
segmenter.process(image);
```

上面的代码会修改标记图像，每个值为0的像素都会被赋予一个输入标签，而边缘处的像素赋值为-1。得到的标签图像如下所示：



边缘图像差不多是这样的：



5.5.2　实现原理

跟前面几节一样，我们在描述分水岭算法时用拓扑地图来做类比。用分水岭算法分割图像的原理是从高度0开始逐步用洪水淹没图像。当“水”的高度逐步增加时（到1、2、3等），会形成聚水的盆地。随着盆地面积逐步变大，两个不同盆地的水最终会汇合到一起。这时就要创建一个分水岭，用来分割这两个盆地。当水位达到最大高度时，创建的盆地和分水岭就组成了分水岭分割图。

可以想象，在水淹过程的开始阶段会创建很多细小的独立盆地。当所有盆地汇合时，就会创建很多分水岭线条，导致图像被过度分割。要解决这个问题，就要对这个算法进行修改，使得水淹的过程从一组预先定义好的标记像素开始。每个用标记创建的盆地，都按照初始标记的值加上标签。如果两个标签相同的盆地汇合，就不创建分水岭，以避免过度分割。调用`cv::watershed`函数时就执行了这些过程。输入的标记图像会被修改，用以生成最终的分水岭分割图。输入的标记图像可以含有任意数值的标签，未知标签的像素值为`0`。标记图像的类型选用32位有符号整数，以便定义超过255个的标签。另外，可以把分水岭的对应像素设为特殊值`-1`。这是由`cv::watershed`函数返回的。

为了方便显示结果，我们采用两种特殊方法。第一种方法返回由标签组成的图像（包含值为`0`的分水岭）。该方法通过阈值化很容易地实现，如下：

```
// 以图像的形式返回结果
cv::Mat getSegmentation() {

 cv::Mat tmp;
 // 所有标签值大于255的区段都赋值为255
 markers.convertTo(tmp,CV_8U);

 return tmp;
}
```

类似地，第二种方法返回一个图像，图像中分水岭线条赋值为0，其他部分赋值为255。这次用cv::convertTo方法来获得结果，如下：

```
// 以图像的形式返回分水岭
cv::Mat getWatersheds() {

 cv::Mat tmp;
 // 在变换前，把每个像素p转换为255p + 255
 markers.convertTo(tmp,CV_8U,255,255);

 return tmp;
}
```

在变换前对图像做线性转换，使值为-1的像素变为0（因为–1 * 255 + 255 = 0）。

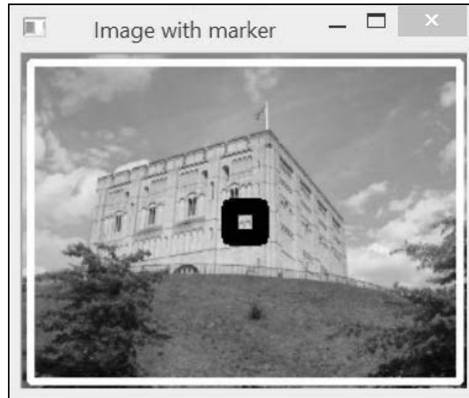
值大于255的像素赋值为255。这是因为有符号整数转换成无符号字符型时，应用了饱和度运算。

5.5.3　扩展阅读

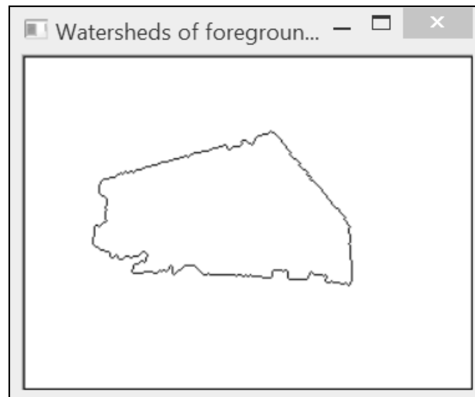
很明显，可以用多种方法获得标记图像。例如，用户可以交互式地在场景中的物体和背景上绘制区域。或者，当需要标识的物体位于图像中间时，可以简单地在输入图像的中心位置标记特定标签，在图像边缘位置（假设背景在边缘位置）标记上另一个标签。可以用下面的方法创建标记图像：

```
// 标识背景像素
cv::Mat imageMask(image.size(),CV_8U,cv::Scalar(0));
cv::rectangle(imageMask,cv::Point(5,5),
     cv::Point(image.cols-5,
               image.rows-5),cv::Scalar(255),3);
//  标识前景像素
// （在图像的中心）
cv::rectangle(imageMask,
     cv::Point(image.cols/2-10,image.rows/2-10),
     cv::Point(image.cols/2+10,image.rows/2+10),
     cv::Scalar(1),10);
```

如果把这个标记图像叠加到实验图像上，将得到下面的图像:



这个图像是分水岭算法得到的结果：



5.5.4　参阅

- C. Vachier和F. Meyer发表在*Journal of Mathematical Imaging and Vision* 2005年5月第22卷第2期和第3期上的文章“The viscous watershed transform”提供了有关于分水岭转换的更多信息；
- 5.7节介绍了另一种图像分割算法，也能把图像分割为背景和前景。

5.6　用 MSER 算法提取特征区域

上一节我们学习了如何通过逐步水淹并创建分水岭，把图像分割成多个区域。最大稳定极值区域（MSER）算法也用相同的水淹类比，以便从图像中提取有意义的区域。创建这些区域时也使用逐步提高水位的方法，但是这次我们关注的是在水淹过程中的某段时间内保持相对稳定的盆地。可以发现，这些区域对应着图像中某些物体的独特部分。

5.6.1 如何实现

计算图像MSER的基础类是cv::MSER。可以用默认的无参数构造函数创建这个类的实例。这里我们通过指定被检测区域的最小和最大尺寸对其进行初始化，以便限制它们的数量。因此调用方式如下：

```
// 基本的MSER检测器
cv::MSER mser(5, // 检测极值区域时使用的增量
            200, // 允许的最小面积
          1500); // 允许的最大面积
```

现在，可以通过调用一个仿函数来获得MSER，指定输入图像和一个相关的输出数据结构，代码如下：

```
// 点集的容器
std::vector<std::vector<cv::Point>> points;
// 检测MSER特征
mser(image, points);
```

得到的结果是一个包含若干个区域容器，每个区域用组成它的像素点表示。为了呈现结果，我们创建一个空白图像，在图像上用不同的颜色显示检测到的区域（颜色是随机选择的）。用以下代码实现：

```
// 创建白色图像
cv::Mat output(image.size(),CV_8UC3);
output= cv::Scalar(255,255,255);

// 随机数生成器
cv::RNG rng;

// 针对每个检测到的特征区域
for (std::vector<std::vector<cv::Point>>::
          iterator it= points.begin();
       it!= points.end(); ++it) {

  // 生成随机颜色
  cv::Vec3b c(rng.uniform(0,255),
             rng.uniform(0,255),
             rng.uniform(0,255));

  // 针对MSER集合中的每个点
  for (std::vector<cv::Point>::
               iterator itPts= it->begin();
            itPts!= it->end(); ++itPts) {

    //不重写MSER的像素
    if (output.at<cv::Vec3b>(*itPts)[0]==255) {

      output.at<cv::Vec3b>(*itPts)= c;
    }
  }
}
```

注意，MSER会形成层叠区域。为了显示全部区域，不能重写大区域内包含的小区域。如果从下图检测出MSER：



那么，最后显示的图像如下（另见彩图8）：



这个检测结果并不粗糙。但是可以看出，通过这种方法能从图片中提取到一些有意义的区域（例如建筑物的窗户）。

5.6.2　实现原理

MSER使用的原理与分水岭算法相同；即高度为0~255，逐渐淹没图像。随着水位的增高，被严格界定黑色区域会形成盆地，并且会在一段时间内有相对稳定的形状（在水淹类比下，水位高低代表了像素值的强度）。这些稳定的盆地就是MSER。检测方法是，观察每个水位连通的区域并测量它们的稳定性。而测量稳定性的方法则是比较当前区域和水位上升前的区域的面积。如果相对变化达到局部最小值，就认为这个区域是MSER。增量值将作为`cv::MSER`类构造函数的第一个参数，用以测量相对稳定性，其默认值为`5`。另外，要注意，区域面积必须在预定义的范

围内。构造函数中后面两个参数，就是允许的最小和最大区域尺寸。另外必须确保MSER是稳定的（第四个参数），即形状的相对变化必须足够小。一个稳定区域可以属于另一个更大的区域（称为父区域）。

为了确保有效性，一个父MSER和它的子区域必须有足够大的差别，即差异限度。差异限度由cv::MSER类构造函数的第五个参数指定。在前面的例子中，最后两个参数都使用了默认值。（MSER允许的最大相对变化的默认值为0.25，父MSER与子区域的最小差别的默认值为0.2。）

MSER检测的结果是一个包含点集的容器。由于我们通常更关心区域的整体而不是单个像素的位置，因此普遍采用含有位置和大小信息的单一的几何形状来表示MSER。常用的形状是带边缘的椭圆。可通过OpenCV的两个实用函数获得这些椭圆。第一个是cv::minAreaRect函数，它会寻找包含集合中所有像素点的面积最小的矩形。这个矩形可用cv::RotateRect类的实例表示。找到这个带边框的矩形后，就可以用cv::ellipse函数在图像上画出内切椭圆。我们把这个完整的过程封装进一个类，这个类的构造函数和cv::MSER类的构造函数基本相同。参见以下代码：


```
class MSERFeatures {

  private:

    cv::MSER mser;        // MSER检测器
    double minAreaRatio; // 额外的排斥参数

  public:

    MSERFeatures(
           // 允许的尺寸范围
           int minArea=60, int maxArea=14400,
           // MSER面积/带边框矩形面积之比的最小值
           double minAreaRatio=0.5,
           // 用以测量稳定性的增量值
           int delta=5,
           // 最大允许面积变化量
           double maxVariation=0.25,
           // 子区域和父区域之间差距的最小值
           double minDiversity=0.2):
           mser(delta,minArea,maxArea,
           maxVariation,minDiversity),
           minAreaRatio(minAreaRatio) {}
```

加入了一个额外的参数（minAreaRatio），当MSER的面积与它对应的带边框矩形的面积差别太大时，就可以使用这个参数把MSER消除。这样可清理掉不需要太关注的细长形状。

用下面的方法计算代表MSER的带边框矩形的列表：

```
// 得到对应每个MSER特征的旋转带边框的矩形
// 如果(MSER面积 / 矩形面积) < areaRatio，就清除这个特征
```

```
void getBoundingRects(const cv::Mat &image,
                      std::vector<cv::RotatedRect> &rects) {

  // 检测MSER特征
  std::vector<std::vector<cv::Point>> points;
  mser(image, points);

  // 针对每个检测到的特征
  for (std::vector<std::vector<cv::Point>>::
            iterator it= points.begin();
         it!= points.end(); ++it) {

        // 提取带边框的矩形
        cv::RotatedRect rr= cv::minAreaRect(*it);

        // 检查面积比例
        if (it->size() > minAreaRatio*rr.size.area())
        rects.push_back(rr);
  }
}
```

用下面的方法在图像上画出对应的椭圆：

```
// 画出对应每个MSER的旋转椭圆
cv::Mat getImageOfEllipses(const cv::Mat &image,
            std::vector<cv::RotatedRect> &rects,
            cv::Scalar color=255) {

  // 画到这个图像上
  cv::Mat output= image.clone();

  // 得到MSER特征
  getBoundingRects(image, rects);

  // 针对每个检测到的特征
  for (std::vector<cv::RotatedRect>::
            iterator it= rects.begin();
         it!= rects.end(); ++it) {

        cv::ellipse(output,*it,color);
  }

  return output;
}
```

然后用以下代码检测MSER：

```
// 创建MSER特征检测器的实例
MSERFeatures mserF(200,   // 最小面积
                  1500,   // 最大面积
                  0.5);   // 面积比率阈值
                          // 使用默认增量值

// 存放带边框的旋转矩形的容器
```

```
std::vector<cv::RotatedRect> rects;

// 检测并取得图像
cv::Mat result= mserF.getImageOfEllipses(image,rects);
```

将这个函数应用到前面用到的图像中，得到的图像如下：



将这个结果和前面的结果做比较，就会发现后一种展现方式更容易解读。注意有的子MSER和父MSER的椭圆非常接近。这种情况下，可以针对椭圆使用一个最小变化限值，以便清理掉重复的椭圆。

5.6.3 参阅

- 7.6节将介绍计算连通点集的其他属性的方法。
- 第8章将解释如何把MSER作为兴趣点检测器。

5.7 用 GrabCut 算法提取前景物体

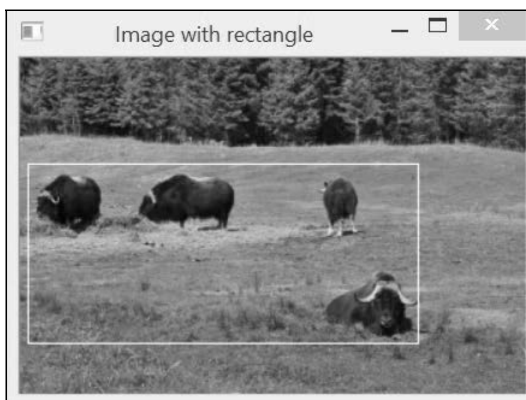
OpenCV实现了另一种常用的图像分割算法：GranCut算法。它并不是基于数学形态学的，但是在用法上与前面的分水岭分割算法比较相似，因此把它放在这里讲述。GrabCut的计算速度比分水岭算法要慢，但是得到的结果通常更精确。如果要从静态图像中提取前景物体（例如从一个图像剪切物体粘贴到另一个图像），采用GrabCut算法是最好的选择。

5.7.1 如何实现

`cv::grabCut`函数的用法非常简单。只需要输入一个图像，并对一些像素做上属于背景或属于前景的标记。 算法会根据这个局部的标记，计算出整个图像中前景/背景的分割线。

为输入图像指定局部的前景/背景标签，方法之一就是定义一个包含前景物体的矩形：

```
// 定义一个带边框的矩形
// 矩形外部的像素会被标记为背景
cv::Rect rectangle(5,70,260,120);
```



矩形外部的像素都会被标记为背景。调用cv::grabCut时，除了需要输入图像和分割后的图像，还需要定义两个矩阵，用于存放算法构建的模型，代码如下：

```
cv::Mat result; // 分割结果（4种可能的值）
cv::Mat bgModel,fgModel; // 模型（内部使用）
// GrabCut分割算法
cv::grabCut(image,     // 输入图像
          result,      // 分割结果
          rectangle,   // 包含前景的矩形
          bgModel,fgModel, // 模型
          5,           // 迭代次数
          cv::GC_INIT_WITH_RECT); // 使用矩形
```

注意，我们在函数的中用cv::GC_INIT_WITH_RECT标志作为最后一个参数，表示将使用带边框的矩形模型（后面会讨论其他模式）。输入/输出的分割图像可以是以下四个值之一。

- cv::GC_BGD：这个值表示明确属于背景的像素（例如，本例中矩形以外的像素）。
- cv::GC_FGD：这个值表示明确属于前景的像素（本例中没有这种像素）。
- cv::GC_PR_BGD：这个值表示可能属于背景的像素。
- cv::GC_PR_FGD：这个值表示可能属于前景的像素（即本例中矩形内像素的初始值）。

通过提取值为cv::GC_PR_FGD的像素，可得到含分割信息的二值图像。参见以下代码：

```
// 取得标记为“可能属于前景”的像素
cv::compare(result,cv::GC_PR_FGD,result,cv::CMP_EQ);
// 生成输出图像
cv::Mat foreground(image.size(),CV_8UC3,
                   cv::Scalar(255,255,255));
image.copyTo(foreground,// 不复制背景像素
```

```
                result);
```

提取所有前景像素，即值为cv::GC_FGD或cv::GC_PR_FGD的像素。可以检查数值的第一位，代码如下：

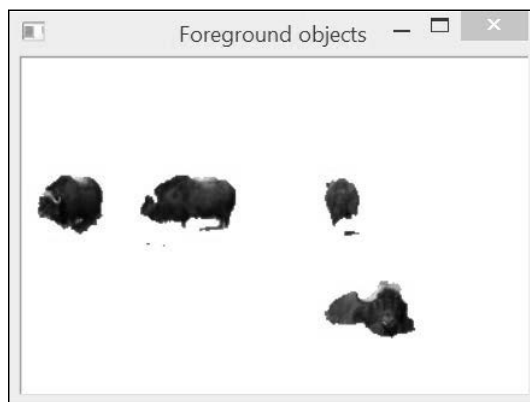
```
// 用按位与运算检查第一位
result= result&1; // 如果是前景像素，结果为1
```

这是因为，这几个常量定义的值为1和3，而另外两个（cv::GC_BGD和cv::GC_PR_BGD）定义为0和2。本例中，因为分割图像不含cv::GC_FGD像素（只输入了cv::GC_BGD像素），所以得到的结果是一样的。

最后，用以下带掩码的复制操作得到前景物体的图像（在白色背景上）：

```
// 生成输出图像
cv::Mat foreground(image.size(),CV_8UC3,
          cv::Scalar(255,255,255)); // 全部为白色的图像
image.copyTo(foreground,result); // 不复制背景像素
```

得到的结果是如下图像：



5.7.2　实现原理

在前面的例子中，只需要指定一个包含前景物体的矩形，GrabCut算法就能提取出它们。此外，还可以把分割图像中的几个特定像素赋值为cv::GC_BGD和cv::GC_FGD，并把这个分割图像作为cv::grabCut函数的第二个参数。然后在输入模式中指定GC_INIT_WITH_MASK。有多种方式可获得这些输入标签，例如提示用户在图像中交互式地标记一些元素。也可以结合使用这两种输入模式。

利用这个输入信息，GrabCut算法用以下步骤进行背景/前景分割。首先，把所有未标记的像素临时标为前景（cv::GC_PR_FGD）。基于当前的分类情况，算法把像素划分为多个颜色相似的

组（即K个背景组和K个前景组）。下一步是通过引入前景和背景像素之间的边缘，来确定背景/前景的分割。这将通过一个优化过程来实现，过程中试图连接具有相似标记的像素，并且避免在强度相对一致的区域上放置边缘。使用**Graph Cuts**算法可以高效地解决这个优化问题。此算法用这种方式来寻找最优的解决方案：把问题表示成一副连通的图形，然后在图形上进行切割，以形成最优的形态。分割完成后，像素会有新的标记。然后重复这个分组过程，找到新的最优分割方案，如此反复。因此，GrabCut算法是一个逐步改进分割结果的迭代过程。根据场景的复杂程度，找到最佳方案所需的迭代次数各不相同（对于简单的情况，迭代一次就足够了）。

这解释了函数中用来表示迭代次数的参数。算法内部用到了两个模型，它们被作为函数的参数传入（并返回）。因此，如果希望通过执行额外的迭代过程来改进分割结果，可以在调用函数时重复使用上次运行的模型。

5.7.3　参阅

- C. Rother、V. Kolmogorov和A. Blake发表在*ACM Transactions on Graphics (SIGGRAPH)* 2004年8月第23卷第3期上的文章“GrabCut: Interactive Foreground Extraction using Iterated Graph Cuts”详细描述了GrabCut算法。

第 6 章

图像滤波

本章包括以下内容：

- ❑ 用低通滤波器进行图像滤波；
- ❑ 用中值滤波器进行图像滤波；
- ❑ 用定向滤波器检测边缘；
- ❑ 计算图像的拉普拉斯算子。

6.1 简介

滤波是信号和图像处理中的一种基本操作。它的目的是选择性地提取图像中某些方面的内容，这些内容在特定应用环境下传达了重要信息。滤波可去除图像中的噪声，提取有用的视觉特征，对图像重新采样，等等。它起源于通用的*信号和系统*理论。这里不对这个理论作详细解释。本章将介绍几个有关滤波的重要概念，并演示如何在图像处理程序中使用滤波器。首先简要解释一下频域分析的概念。

当观察一幅图像时，我们看到不同的灰度级（或颜色）在图像上的分布状况。图像之间的区别，就在于它们有不同的灰度级分布方式。然而，也可以从其他角度进行图像分析。我们可以看到图像中灰度级的变化。有些图像含有大片强度值几乎不变的区域（如蓝天），而对于其他图像，灰度级的强度值在整幅图像上的变化很大（例如由大量细小物体构成的混乱场景）。因此产生了另一种描述图像特性的方式，即观察上述变化的频率。这种特征称为*频域*（frequency domain）；而通过观察灰度分布来描述图像特征，称为空域（spatial domain）。

频域分析把图像分解成从低频到高频的频率成分。图像强度值变化慢的区域只包含低频率，而强度值快速变化的区域产生高频率。有几种著名的变换法可用来清楚地显示图像的频率成分，例如傅里叶变换或余弦变换。图像是二维的，因此频率分为两种，即垂直频率（垂直方向的变化）和水平频率（水平方向的变化）。

在频域分析框架下，*滤波器*是一种放大图像中某些频段，同时滤掉（或减弱）其他频段的算子。低通滤波器的作用是消除图像中高频部分；高通滤波器刚好相反，其作用是消除图像中低频

部分。本章将介绍几种在图像处理领域常用的滤波器，并解释它们对图像所起的作用。

6.2　低通滤波器

本节将介绍几种非常基本的低通滤波器。在本章的简介部分，我们已经知道这种滤波器目的是减少图像变化的幅度。要做到这点，一个简单的方法是把每个像素的值替换成它周围像素的平均值。这样一来，强度的快速变化会被消除，代之以更加平滑的过渡。

6.2.1　如何实现

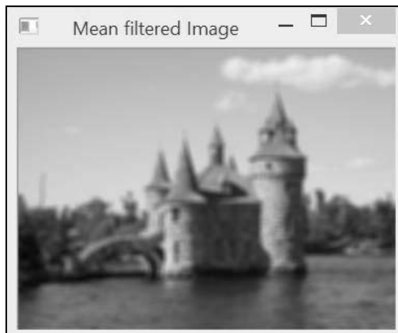
cv::blur函数将每个像素的值替换成该像素邻域的平均值（邻域是矩形的），从而使图像更加平滑。这个低通滤波器的用法如下：

```
cv::blur(image,result,
         cv::Size(5,5)); // 滤波器尺寸
```

这种滤波器也称为块滤波器（box filter）。为了让效果更加明显，这里我们使用尺寸为5 × 5的滤波器。先看下面的图像：



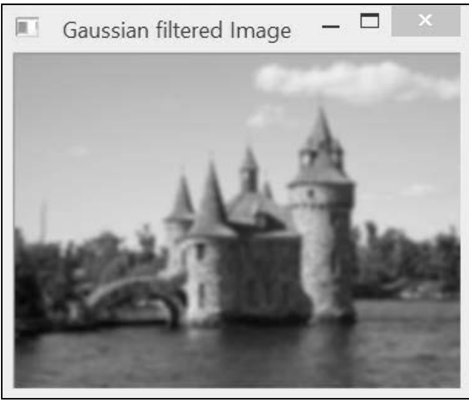
使用滤波器后得到的结果如下：



有时需要让邻域内较近的像素具有更高的重要度。因此可计算加权平均值，即较近的像素比较远的像素具有更大的权重。要得到加权平均值，可采用依据高斯函数（即“钟形曲线”函数）制定的加权策略。函数cv::GaussianBlur应用了这种滤波器，调用方法如下：

```
cv::GaussianBlur(image,
        result, cv::Size(5,5), // 滤波器尺寸
        1.5);   // 控制高斯曲线形状的参数
```

得到的结果如下：



6.2.2 实现原理

如果用邻域像素的加权累加值来替换像素值，我们就说这种滤波器是线性的。这里使用了均值滤波器，即将矩形邻域内的全部像素累加，除以该邻域的数量（即求平均值），然后用这个平均值替换原像素的值。这相当于把邻域中每个像素乘以1，然后进行累加。也可以把邻域中每个像素位置对应的放大系数存放在一个矩阵中，用这个矩阵表示滤波器的不同权重。矩阵中心的元素对应当前正在应用滤波器的像素。这样的矩阵也称为*内核*或*掩码*。对于一个3 × 3均值滤波器，其对应的内核可能是这样的：

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

函数cv::boxFilter对图像做滤波时，使用了一个仅由1组成的正方形内核。它与均值滤波器类似，但不会除以系数的数量。

应用一个线性滤波器，相当于将内核移动到图像的每个像素上，并将每个对应像素乘以它的权重。这个运算在数学上称为*卷积*，规范的写法如下：

$$I_{out}(x,y) = \sum_i \sum_j I_{in}(x-i, y-j)K(i,j)$$

在这个双重求和过程中，位于（x,y）的当前像素与K内核的中心点对齐，并假定它位于坐标（0, 0）处。

观察本节产生的输出图像，可以发现低通滤波器的最终效果是使图像更加模糊或更加平滑。这不奇怪,因为低通滤波器减弱了高频成分,而高频成分正好对应了物体边缘处的快速视觉变化。

对于高斯滤波器，像素对应的权重与它到中心像素之间的距离成正比。一维高斯函数的公式为：

$$G(x) = A\mathrm{e}^{-x^2/2\sigma^2}$$

使用归一化系数A，是为了确保各个权重的累加和等于1。符号σ（sigma，希腊字母西格玛）的值决定了高斯函数曲线的宽度。这个值越大，函数曲线就越扁平。例如计算一维高斯滤波器的系数，区间[–4, 0, 4]，如果$\sigma = 0.5$，得到如下系数：

```
[0.0 0.0 0.00026 0.10645 0.78657 0.10645 0.00026 0.0 0.0]
```

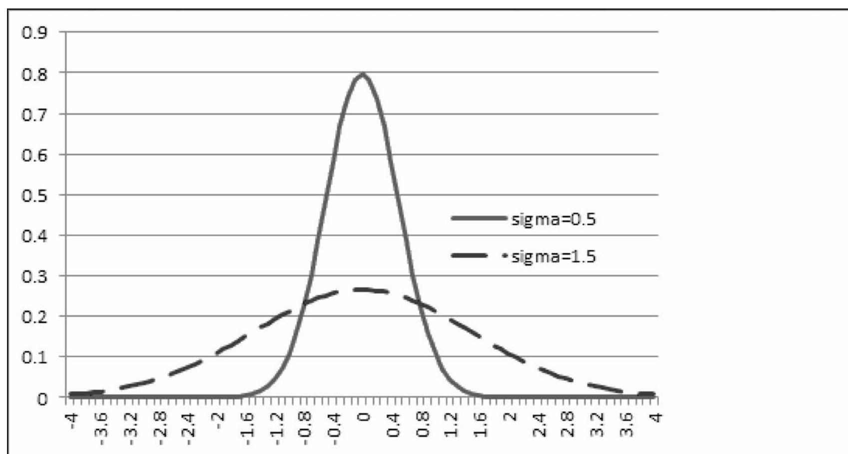
如果$\sigma = 1.5$，得到的系数为：

```
[0.00761 0.036075 0.10959 0.21345 0.26666
 0.21345 0.10959 0.03608 0.00761 ]
```

注意，计算这些系数的方法是用对应的σ的值，调用函数`cv::getGaussianKernel`：

```
cv::Mat gauss= cv::getGaussianKernel(9, sigma,CV_32F);
```

高斯函数是一个对称钟形曲线，这使它非常适合用于滤波。参见下图：



离中心点越远的像素权重越低，这使像素之间的过渡更加平滑。与之相反，使用扁平的均值滤波器时，远处的像素会使当前平均值发生突变。从频率上看，这意味着均值滤波器并没有消除全部高频成分。

要在图像上应用二维高斯滤波器，只需先在横向线条上应用一维高斯滤波器（过滤水平方向的频率），然后在纵向线条上应用另一个一维高斯滤波器（过滤垂直方向的频率）。这是因为，高斯滤波器是一种可分离的滤波器（也就是说，二维内核可分解成两个一维滤波器）。要应用普通的可分离滤波器，可使用`cv::sepFilter2D`函数。也可以用`cv::filter2D`函数直接应用二维内核。由于可分离滤波器所用的乘法运算更少，因此它的计算速度通常比不可分离滤波器要快。

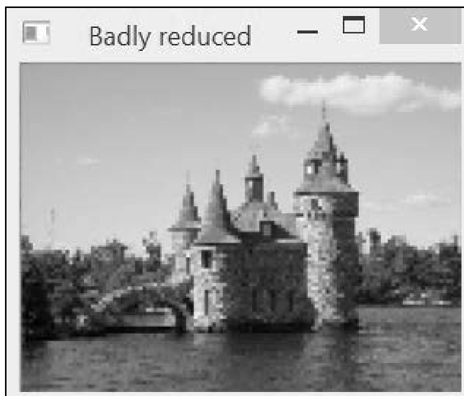
在OpenCV中，要对图像应用高斯滤波器，需要调用函数`cv::GaussianBlur`，并且提供系数的个数（第三个参数，必须是奇数）和σ的值（第四个参数）。也可以只设置σ的值，由OpenCV决定系数的个数（输入滤波器尺寸的值为`0`）。反过来也可以，即输入参数时提供尺寸的数值，σ值为`0`。函数会自行判断最适合尺寸的σ值。

6.2.3 扩展阅读

调整图像大小时也要使用低通滤波器，本节将解释这么做的原因。调整图像大小时还可能需要插入像素值，本节也将讨论这方面内容。

1. 缩减像素采样

也许你会觉得，要缩小一个图像，只需简单地消除图像中的一部分行和列。可惜，这么做得到的图像效果很差。下面的图片说明了这点，它是由测试用的图像缩小到1/4后得到的，缩小的方式是每四行（列）像素中保留一行（列）。注意，为了让图像的缺陷看起来更明显，每个像素按原大小的两倍进行显示，从而放大了图像（下一节将解释如何实现）。参见下面的图片：



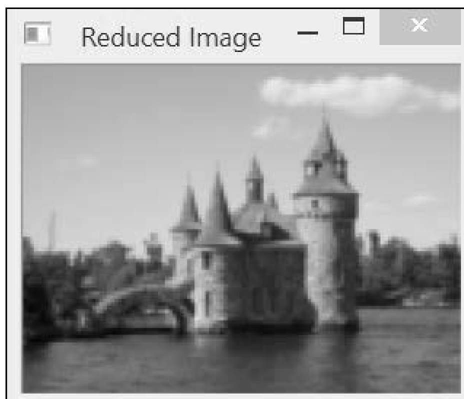
可以发现，这个图像的质量明显降低了。例如原始图像中城堡顶部倾斜的边缘，在缩小后的

图像中看起来像是楼梯。图像的纹理部分也能看到锯齿状的变形（如砖墙）。

这些令人讨厌的伪影是一种叫作*空间假频*的现象造成的。当你试图在图像中包含高频成分，但是图像太小无法包含时，就会出现这种现象。实际上，在小图像（即像素较少的图像）中展现精致纹理和减税边缘，效果不如较高分辨率的图像（想想高清电视机和普通电视机的差别）。图像中精致的细节对应着高频，因此我们需要在缩小图像之前去除它的高频成分。通过学习本节内容，我们知道这可以用低通滤波器实现。因此在删除部分列和行之前，必须首先在原始图像上应用低通滤波器，这样才能使图像在缩小到四分之一后不出现伪影。这是用OpenCV的实现方法：

```
// 首先去除高频成分
cv::GaussianBlur(image,image,cv::Size(11,11),2.0);
// 每4个像素中，只保留1个
cv::Mat reduced2(image.rows/4,image.cols/4,CV_8U);
for (int i=0; i<reduced2.rows; i++)
  for (int j=0; j<reduced2.cols; j++)
    reduced2.at<uchar>(i,j)= image.at<uchar>(i*4,j*4);
```

得到的图像如下：



当然了，这个图像丢失了一些精致的细节，但是总体上看，它的视觉质量比前面的要好。

你也可以用OpenCV的一个专用函数实现图像缩小。即`cv::pyrDown`函数：

```
cv::Mat reducedImage; // 用于存储缩小后的图像
cv::pyrDown(image,reducedImage); // 图像尺寸缩小一半
```

上述函数使用一个5 × 5的高斯滤波器，在把图像缩小一半之前先进行低通滤波。此外还有功能相反的函数`cv::pyrUp`，可以放大图像的尺寸。这种提升像素采样的过程，是先在每两行和两列之间分别插入值为0的像素，然后对扩展后的图像应用同样的5 × 5高斯滤波器（但系数要扩大4倍）。先缩小后放大一个图像，显然不能完全恢复到原始图像。缩小过程中丢失的信息是无法恢复的。这两个函数可用来创建*图像金字塔*。它是一个数据结构，由一个图像不同尺寸的版本堆叠起来（这里每层的图像的尺寸是后一层的2倍，但是这个比例还可以更小，例如1.2），经常用于

高效的图像分析。例如，如果要在图像中检测一个物体，可以首先在金字塔顶部的小图像上检测。当定位到关注的物体时，在金字塔的更低层次进行更精细的搜索，更低层次的图像分辨率更高。

此外还有一个更通用的函数cv::resize，可以指定缩放后图像的尺寸。只需要在调用它时指定新的尺寸，该尺寸可以比原始图像更小，也可以更大：

```
cv::Mat resizedImage; // 用于存储缩放后的图像
cv::resize(image,resizedImage,
     cv::Size(image.cols/4,image.rows/4)); // 行和列均缩小为原来的1/4
```

也可以指定缩放比例。在参数中提供一个空的图像实例，然后提供缩放比例：

```
cv::resize(image,resizedImage,
           cv::Size(), 1.0/4.0, 1.0/4.0); // 缩小为原来的1/4
```

最后一个参数可用来选择重新采样时使用的插值方法。这在下面的段落中介绍。

2. 像素插值

图像按比例缩放后，必须进行像素插值，以便在原像素之间的位置插入新的像素值。通用的图像重映射（见2.8节），属于另一种需要像素插值的情况。

进行插值的最基本方法是使用*最近邻策略*。把待生成图像的像素网格放在原图像的上方，每个新像素被赋予原图像中最邻近像素的值。当图像升采样时（即新网格比原始网格更密集时），意味着新网格中有多于一个的像素会采用原网格中同一个像素的值。

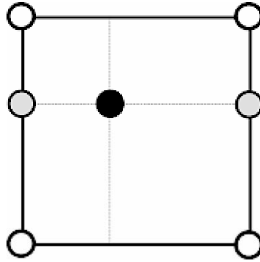
例如要把上面缩小后的图像放大3倍，采用最邻近插值法（通过插值标志cv::INTER_NEAREST实现），代码如下：

```
cv::resize(reduced, newImage,
             cv::Size(), 3, 3,cv::INTER_NEAREST);
```

得到的结果如下：



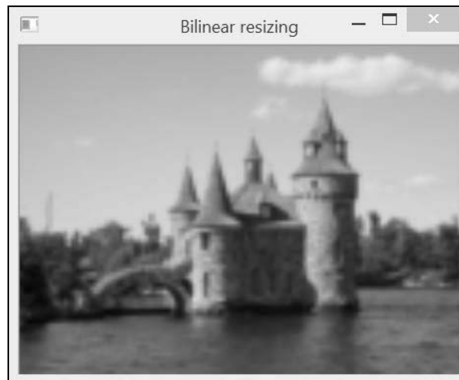
本例中的插值算法简单地把每个像素的尺寸乘以3（这也是上一节生成图像的方法）。更好的做法是在插入新的像素值时，结合多个邻近像素的值。因此，可利用周围四个像素的值线性地计算新像素值，如下图所示：



具体过程为，首先在新增像素的左侧和右侧垂直地插入两个像素值。然后利用这两个插入的像素（上面图片中的灰色部分），在预定的位置水平地插入像素值。这种双线性插值方案是`cv::resize`函数的缺省方法（可以用标志`cv::INTER_LINEAR`显式地指定）：

```
cv::resize(reduced2, newImage,
           cv::Size(), 3, 3, cv::INTER_LINEAR);
```

得到的结果如下：



此外还有一些算法，可以得到更好的结果。使用*双三次插值算法*，在执行插值运算时需要考虑4×4的邻域像素。不过，因为这种算法使用了更多的像素，并且包含了三次函数的计算，所以它的运算速度比双线性插值算法慢。

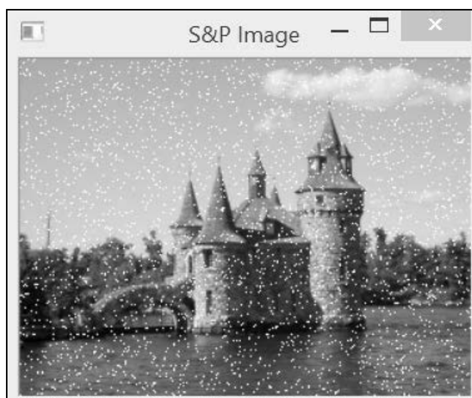
6.2.4　参阅

- 2.6.4节介绍了`cv::filter2D`函数。该函数根据用户选择的内核，在图像上应用线性滤波器。

6.3　中值滤波器

6.2节介绍了线性滤波器的概念。此外，非线性滤波器在图像处理中也起着很重要的作用。本节介绍的中值滤波器就是其中的一种。

因为中值滤波器非常有助于消除椒盐噪声中（这里我们用只有盐的噪声），所以我们将使用2.1节中创建的图像，如下：



6.3.1　如何实现

调用中值滤波器函数的方法，与其他滤波器差不多：

```
cv::medianBlur(image,result,5); // 滤波器尺寸
```

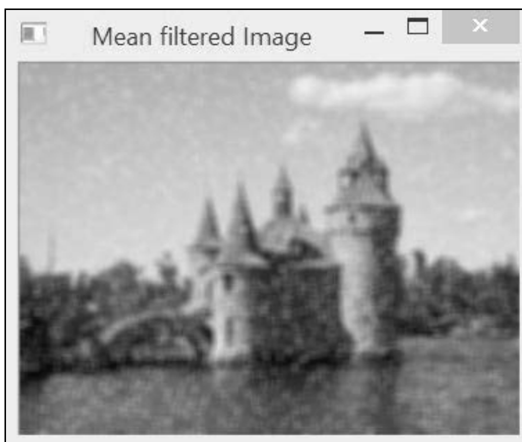
结果如下：



6.3.2　实现原理

因为中值滤波器是非线性的，所以不能用核心矩阵表示。但它也是通过操作一个像素的邻域，来确定输出的像素值。正如它的名称所示，中值滤波器把当前像素和它的领域组成一个集合，然后计算出这个集合的中间值，以此作为当前像素的值（集合中数值经过排序，中间位置的数值就是中间值）。

这正是中值滤波器在消除椒盐噪声中如此高效的原因。事实上，如果在某个邻域中有一个异常的黑色或白色像素，该像素将无法作为中间值（它是最大值或最小值），因此肯定会被邻域的值替换掉。相反，简单均值滤波器会在很大程度上受到这种噪声影响，从下图中可以观察到这种影响，它是用均值滤波器消除椒盐噪声的结果：



很明显，包含噪声的像素使邻域的均值发生了偏移。虽然噪声被均值滤波器模糊了，但仍然可以看见。

中值滤波器还有利于保留边缘的尖锐度。但是它会洗去均质区域中的纹理（例如背景中的树木）。因为中值滤波器具有良好的视觉效果，因此照片编辑软件常用它创建特效。可用彩色图像来测试，看它如何生成类似卡通的图像。

6.4　用定向滤波器检测边缘

6.2节介绍了用核心矩阵进行线性滤波的概念。这些滤波器通过移除或减弱高频成分，达到模糊图像的效果。本节我们执行一种反向的变换，即放大图像中的高频成分。然后用本节介绍的高通滤波器进行边缘检测。

6.4.1 如何实现

我们将要使用的滤波器称为**Sobel**滤波器。因为它只对垂直或水平方向的图像频率起作用（具体方向取决于滤波器选用的内核），所以它被认为是一种定向滤波器。OpenCV中有一个函数可在图像上应用Sobel算子。水平方向滤波器的调用方法为：

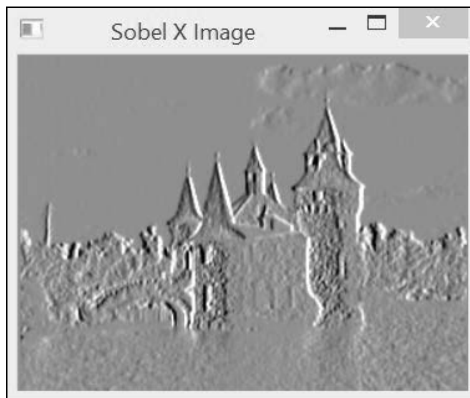
```
cv::Sobel(image,    // 输入
          sobelX,    // 输出
          CV_8U,     // 图像类型
          1, 0,      // 内核规格
          3,         // 正方形内核的尺寸
          0.4, 128); // 比例和偏移量
```

垂直方向滤波的调用方法为（与水平方向滤波器非常类似）：

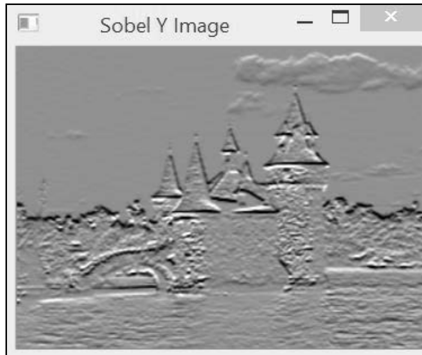
```
cv::Sobel(image,    // 输入
          sobelY,    // 输出
          CV_8U,     // 图像类型
          0, 1,      // 内核规格
          3,         // 正方形内核的尺寸
          0.4, 128); // 比例和偏移量
```

函数用到了几个整型参数，下一节会详细解释。注意选用这些参数是为了生成一个8位的输出图像（`CV_8U`）。

水平方向Sobel算子得到的结果如下：



下一节我们将看到，Sobel算子的内核中有正数也有负数，因此Sobel滤波器的计算结果通常是16位的有符号整数图像（`CV_16S`）。为了把结果显示为8位图像（上图），我们用数值0代表灰度128。负数表示更黑的像素，正数表示更亮的像素。垂直方向Sobel图像如下：



如果你熟悉照片编辑软件，就会知道这像是图像浮雕化特效，实际上这种特效通常就是用定向滤波器生成的。

你可以组合这两个结果（垂直和水平方向），得到Sobel滤波器的模：

```
// 计算Sobel滤波器的模
cv::Sobel(image,sobelX,CV_16S,1,0);
cv::Sobel(image,sobelY,CV_16S,0,1);
cv::Mat sobel;
// 计算L1模
sobel= abs(sobelX)+abs(sobelY);
```

在convertTo方法中使用可选的缩放参数，可得到一个图像，图像中白色用0表示，更黑的灰色阴影用大于0的值表示。这个图像可以很方便地显示Sobel算子的模。代码如下：

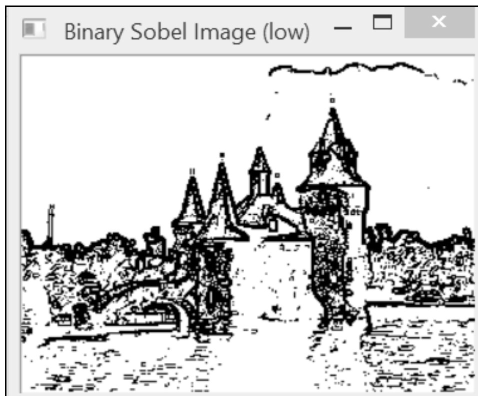
```
// 找到Sobel最大值
double sobmin, sobmax;
cv::minMaxLoc(sobel,&sobmin,&sobmax);
// 转换成8位图像
// sobelImage = -alpha*sobel + 255
cv::Mat sobelImage;
sobel.convertTo(sobelImage,CV_8U,-255./sobmax,255);
```

得到的结果如下：



从这个图像可以看出把这些算子称作边缘检测器的原因。接着可以对这个图像阈值化，得到图像轮廓的二值分布图。代码片段和生成的图像如下：

```
cv::threshold(sobelImage, sobelThresholded,
                      threshold, 255, cv::THRESH_BINARY);
```



6.4.2 实现原理

Sobel算子是一种典型的用于边缘检测的线性滤波器，它基于两个简单的3 × 3内核，内核结构如下：

−1	0	1
−2	0	2
−1	0	1

−1	−2	−1
0	0	0
1	2	1

如果把图像看作二维函数，那么Sobel算子就是图像在垂直和水平方向变化的速度。在数学术语中，这种速度称为梯度。它是一个二维向量，向量的元素是横竖两个方向的函数的一阶导数：

$$grad(I) = \left[\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right]^T$$

Sobel算子在水平和垂直方向计算像素值的差分，得到图像梯度的近似值。它在像素周围的一定范围内进行运算，以减少噪声带来的影响。`cv::Sobel`函数使用Sobel内核来计算图像的卷积。函数的完整说明如下：

```
cv::Sobel(image,  // 输入
          sobel,  // 输出
          image_depth,    // 图像类型
          xorder,yorder,  // 内核规格
          kernel_size,    // 正方形内核的尺寸
          alpha, beta);   // 比例和偏移量
```

输出图像的像素类型是可以选择的，包括无符号字符型、有符号整数或浮点数。如果结果超出了像素值域的范围，就会进行饱和度运算。函数的最后两个参数可用来处理这种情况。在生成最终图像之前，可以将结果缩放（相乘）alpha倍，并加上偏移量beta。前面我们生成的图像中，Sobel值0代表灰度值128（中等灰度），就是用了这种方法。每个Sobel掩码就是一个方向上的导数。因此要用两个参数来指明将要应用的内核，即x方向和y方向导数的阶数。例如，如果xorder和yorder分别为1和0，则得到水平方向Sobel内核；如果分别是0和1，则得到垂直方向的内核。也可以使用其他组合，但这两种组合是最常用的（下节讨论二阶导数的情况）。最后，内核的尺寸也可以大于3 × 3。可选的尺寸有1、3、5和7。内核尺寸为1，表示一维Sobel滤波器（1 × 3或3 × 1）。大尺寸内核的作用，参见6.4.3节。

梯度是一个二维向量，因此它具有模和方向。梯度向量的模表示变化的振幅，计算时通常被当作欧几里德模（也称L2模）：

$$|grad(I)|=\sqrt{\left(\frac{\partial I}{\partial x}\right)^2+\left(\frac{\partial I}{\partial y}\right)^2}$$

但是在图像处理领域，通常把绝对值之和作为模进行计算。这称为L1模，它得到的结果与L2模比较接近，但计算速度快。本节我们采用L1模：

```
// 计算L1模
sobel= abs(sobelX)+abs(sobelY);
```

梯度向量总是指向变化最剧烈的方向。对于一个图像来说，这意味着梯度的方向与边缘垂直，从较黑区域指向较亮区域。梯度的角度用下面的公式计算：

$$\angle grad(I)=\arctan\left(-\frac{\partial I}{\partial y}\Big/\frac{\partial I}{\partial x}\right)$$

在检测边缘时，通常只计算模。但是如果需要同时计算模和方向，可以使用下面的OpenCV函数：

```
// 计算Sobel算子，必须用浮点数类型
cv::Sobel(image,sobelX,CV_32F,1,0);
cv::Sobel(image,sobelY,CV_32F,0,1);
// 计算梯度的L2模和方向
cv::Mat norm, dir;
cv::cartToPolar(sobelX,sobelY,norm,dir);
```

默认情况下，得到的方向用弧度表示。如果要使用角度，只需要增加一个参数并设为`true`。

对梯度幅值进行阈值化，可得到一个二值边缘分布图。选择合适的阈值并不容易。如果阈值太低，就会保留太多（厚）的边缘；而如果选用更严格（高）的阈值，就会留下断裂的边缘。为了说明这种需要作出取舍的情况，下面用更高的阈值得到二值边缘分布图并把它与前面的图做对比：



要同时得到较低阈值和较高阈值的优点，有一个办法就是使用滞后阈值化的概念。下一章在介绍Canny算子时将对此进行解释。

6

6.4.3 扩展阅读

还有一些其他的梯度算子，这里我们介绍其中的几个。还可以在应用导数滤波器之前应用高斯平滑滤波器，这会减少对噪声的敏感度，后面会详细解释。

1. 梯度算子

Prewitt算子定义了下面的内核，用来计算某个像素位置的梯度：

−1	0	1
−1	0	1
−1	0	1

−1	−1	−1
0	0	0
1	1	1

Roberts算子基于这些简单的2×2内核：

1	0
0	−1

0	1
−1	0

如果要更精确地计算梯度方向，可采用Scharr算子：

−3	0	3
−10	0	10
−3	0	3

−3	−10	−3
0	0	0
3	10	3

注意，你可以在`cv::Sobel`函数中使用Scharr内核，参数为`CV_SCHARR`：

```
cv::Sobel(image,sobelX,CV_16S,1,0, CV_SCHARR);
```

也可以调用`cv::Scharr`函数，效果是一样的：

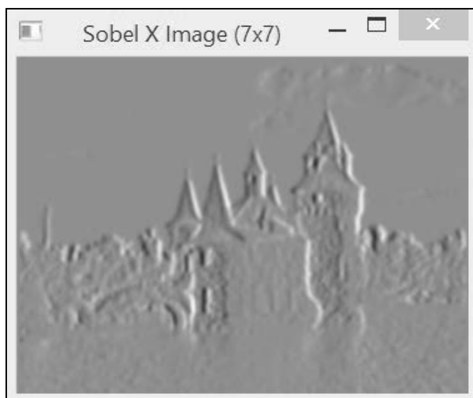
```
cv::Scharr(image,scharrX,CV_16S,1,0,3);
```

所有这些定向滤波器都会计算图像函数的一阶导数。因此，在滤波器方向上像素强度变化大的区域，得到较大的值；较平坦的区域得到较小的值。正因为如此，计算图像导数的滤波器被称为高通滤波器。

2. 高斯导数

导数滤波器属于高通滤波器。因此它们往往会放大图像中的噪声和细小的高对比度细节。为了减少这些高频成分的影响，最好在应用导数滤波器之前对图像做平滑化处理。也许你觉得这需要两个步骤，即平滑化图像和计算导数。但仔细观察这些运算后就能发现，只要选用合适的平滑内核，这两个步骤是可以合并的。前面我们学过，图像与滤波器的卷积可以表示为一些项的累加和。有趣的是，有一个著名的数学定理：项的累加和的导数，等于项的导数的累加和。

因此，可以先对内核求导数，然后与图像卷积，而不是对平滑化的结果求导数。因为高斯内核是连续可导的，所以这种做法特别合适。用不同尺寸的内核调用`cv::Sobel`函数时，就采用了这种方法。这个函数用不同的σ值计算高斯可导内核。例如，如果在x方向选用7 × 7的Sobel滤波器（即`kernel_size=7`），将会得到以下结果：



将它与前面的图像比较，可以发现很多精致的细节已经被移除，明显的边缘位置得到了进一步强化。注意，这时它已经成为一个带通滤波器，较高的频率被高斯滤波器移除，同时较低的频率被Sobel滤波器移除。

6.4.4 参阅

- 7.2节将介绍如何用两个不同的阈值获得二值边缘分布图。

6.5 计算拉普拉斯算子

拉普拉斯算子也是一种基于图像导数运算的高通线性滤波器。它通过计算二阶导数来度量图像函数的曲率。

6.5.1 如何实现

在OpenCV中，可用`cv::Laplacian`函数计算图像的拉普拉斯算子。它与`cv::Sobel`函数非常类似。实际上，为了获得核心矩阵，它们使用了同一个基本函数`cv::getDerivKernels`。根据定义，它采用二阶导数，因此和`cv::Sobel`函数唯一的区别是它没有用来表示导数的阶的参数。

我们为这个算子创建一个简单的类，封装几个与拉普拉斯算子有关的运算。基本的方法如下：

```
class LaplacianZC {

  private:
    // 拉普拉斯算子
    cv::Mat laplace;
    // 拉普拉斯内核的孔径大小
    int aperture;
```

```
public:

   LaplacianZC() : aperture(3) {}

   // 设置内核的孔径大小
   void setAperture(int a) {
      aperture= a;
   }

   // 计算浮点数类型的拉普拉斯算子
   cv::Mat computeLaplacian(const cv::Mat& image) {

      // 计算拉普拉斯算子
      cv::Laplacian(image,laplace,CV_32F,aperture);
      return laplace;
   }
```

拉普拉斯算子的计算在浮点数类型的图像上进行。与上节一样，要对结果做缩放处理才能使其正常显示。缩放基于拉普拉斯算子的最大绝对值，其中数值0对应灰度级128。类中有一个方法可获得下面的图像表示：

```
// 获得拉普拉斯结果，存在8位图像中
// 0表示灰度级128
// 如果不指定缩放比例，那么最大值会放大到255
// 在调用这个函数之前，必须先调用computeLaplacian
cv::Mat getLaplacianImage(double scale=-1.0) {
   if (scale<0) {
      double lapmin, lapmax;
      // 取得最小和最大拉普拉斯值
      cv::minMaxLoc(laplace,&lapmin,&lapmax);
      // 缩放拉普拉斯算子到127
      scale= 127/ std::max(-lapmin,lapmax);
   }

   // 生成灰度图像
   cv::Mat laplaceImage;
   laplace.convertTo(laplaceImage,CV_8U,scale,128);
   return laplaceImage;
}
```

使用这个类，从7 × 7内核计算拉普拉斯图像的方法为：

```
// 用LaplacianZC类计算拉普拉斯算子
LaplacianZC laplacian;
laplacian.setAperture(7); // 7×7的拉普拉斯算子
cv::Mat flap= laplacian.computeLaplacian(image);
laplace= laplacian.getLaplacianImage();
```

得到的图像如下：



6.5.2 实现原理

二维函数的拉普拉斯算子的正式定义为：对它的二阶导数求和：

$$laplace(I) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

如采用最简单的形式，它可以近似为这个3 × 3内核：

0	1	0
1	–4	1
0	1	0

相对Sobel算子而言，拉普拉斯算子在计算时可以使用更大的内核，并且对图像噪声更加敏感，因此最好采用拉普拉斯算子（除非要重点考虑计算效率）。因为这些更大的内核是用高斯函数的二阶导数计算的，因此这个算子也常称为*高斯–拉普拉斯算子*（LoG）。注意，拉普拉斯算子内核的值累加和总是等于0。这保证在强度值恒定的区域中，拉普拉斯算子将变为0。因为拉普拉斯算子度量的是图像函数的曲率，所以在平坦区域中它应该等于0。

初看起来，好像很难解释拉普拉斯算子的作用。从内核的定义可以明显地看出，任何孤立的像素值（即与周围像素差别很大的值）都会被拉普拉斯算子放大。这是因为该算子对噪声非常敏感。但更值得关注的是图像边缘附近的拉普拉斯值。图像边缘的出现是灰度值在区域之间快速地过渡的结果。观察图像函数在边缘（例如从暗到亮的边缘）上的变化，可以发现一个规律：如果灰度级上升，那么肯定存在从正曲率（强度值开始上升）到负曲率（强度值即将到达高地）的平缓过渡。因此，如果拉普拉斯值从正数过渡到负数，就说明这个位置很可能是边缘。或者说，边缘位于拉普拉斯函数的*过零点*。为了说明这个观点，我们来看测试图像的一个小窗口中的拉普拉斯值。我们选取城堡塔楼屋顶的底部边缘位置。具体位置见下图中的白色小框：



现在来观察这个窗口内的拉普拉斯值（7 × 7内核），如下图：

30	-128	39	-12	-37	52	-9	0	0	0	0	-1
123	12	-97	-16	38	-43	8	2	0	0	0	0
-59	76	67	-102	-32	15	-5	-8	2	3	0	1
-82	-75	-48	72	-128	-28	10	8	3	-10	4	-2
72	63	43	81	74	-128	-33	-12	1	-2	7	0
16	23	-12	31	127	127	-58	-11	11	-1	-11	2
-3	-22	71	48	-11	-128	-12	-10	0	0	7	-4
3	10	-17	75	-14	-86	-2	5	1	-4	0	0
-1	2	-14	46	-10	-44	1	2	2	4	2	5
-26	-25	16	112	-30	-96	2	-3	-2	-6	-6	-6
-12	-11	6	97	-28	-62	4	1	4	8	9	8
8	-4	9	117	-8	-71	6	3	6	1	3	3

如果仔细追踪拉普拉斯图像的过零点（位于不同符号的像素之间），就可以得到一条曲线，对应图像窗口中可以看到的边缘。在上面的图片中，我们沿着过零点画了一条虚线，它对应着塔楼的边缘，在选中的图像窗口中可以看到这个边缘。这意味着，理论上讲甚至可以检测到子像素级精度的图像边缘。

在拉普拉斯图像上追踪过零点曲线，需要很大的耐心。但是可以用一个简化的算法，来检测过零点的大致位置。这种算法首先对拉普拉斯图像阈值化，采用的阈值为0，得到正数和负数之间的分割区域。这两个区域之间的轮廓就是过零点。所以我们可用形态学运算来提取这些轮廓，也就是拉普拉斯图像减去膨胀后的图像（这是5.4节中介绍的Beucher梯度）。下面的方法实现了这个算法，生成一个过零点的二值图像：

```
// 获得过零点的二值图像
// 拉普拉斯图像的类型必须是CV_32F
cv::Mat getZeroCrossings(cv::Mat laplace) {

  // 阈值为0
  // 负数用黑色
  // 正数用白色
  cv::Mat signImage;
```

```
  cv::threshold(laplace,signImage,0,255,cv::THRESH_BINARY);

  // 把+/-图像转换成CV_8U
  cv::Mat binary;
  signImage.convertTo(binary,CV_8U);

  // 膨胀+/-区域的二值图像
  cv::Mat dilated;
  cv::dilate(binary,dilated,cv::Mat());

  // 返回过零点的轮廓
  return dilated-binary;
}
```

得到的结果是这个二值分布图：



可以看出，拉普拉斯的过零点方法检测了所有的边缘，而不区分强边缘和弱边缘。我们还知道拉普拉斯算子对噪声非常敏感。最后，有些边缘是由于压缩失真产生的。正是由于这些原因，拉普拉斯算子才检测出了那么多的边缘。事实 上，在检测边缘时只会把拉普拉斯算子与其他算子结合起来使用（例如梯度很大的过零点才能确认的边缘）。在第8章我们会看到，在不同比例下检测兴趣点时，拉普拉斯算子和其他二阶算子是非常有用的。

6.5.3 扩展阅读

拉普拉斯算子是一种高通滤波器。可以组合使用多个低通滤波器，近似地模拟出它。但首先我们要用到图像增强的概念，第2章曾讨论过这个概念。

1. 用拉普拉斯算子增强图像的对比度

可以从图像中减去它的拉普拉斯图像，以增强图像的对比度。这就是我们在第2章中用邻域访问扫描图像时用的方法。当时我们用到了这个内核：

0	−1	0
−1	5	−1
0	−1	0

它等于1减去拉普拉斯内核（也就是原始图像减去它的拉普拉斯图像）。

2. 高斯差分

6.2节中的高斯滤波器可提取图像的低频成分。我们知道高斯滤波器过滤的频率范围，取决于参数σ的值，这个参数控制了滤波器的宽度。现在我们用两个不同带宽的高斯滤波器对一个图像做滤波，然后用这两个结果相减，就得到的由较高的频率构成的图像。这些频率被一个滤波器保留，被另一个滤波器丢弃。这种运算称为高斯差分（Difference of Gaussians，DoG），代码如下：

```
cv::GaussianBlur(image,gauss20,cv::Size(),2.0);
cv::GaussianBlur(image,gauss22,cv::Size(),2.2);

// 计算高斯差分
cv::subtract(gauss22, gauss20, dog, cv::Mat(), CV_32F);

// 计算DoG的过零点
zeros= laplacian.getZeroCrossings(dog);
```

另外，我们也计算DoG算子的过零点，得到如下图像：



可以证明，选择合适的σ值后，DoG算子可以很好地模拟LoG滤波器。另外，如果从一个σ值的增长队列中选取连续的数据对，用以计算一系列的高斯差分，就可以得到该图像的尺度空间表示法。这种多尺度表示法非常有用，例如用于检测尺度无关的图像特征，第8章将详细解释。

6.5.4　参阅

- 8.4节使用拉普拉斯算子和DoG来检测尺度无关的特征。

第 7 章

提取直线、轮廓和区域

本章包括以下内容：

- 用Canny算子检测图像轮廓；
- 用霍夫变换检测直线；
- 点集的直线拟合；
- 提取区域的轮廓；
- 计算区域的形状描述子。

7.1 简介

要进行基于内容的图像分析，我们必须从构成图像的像素集中提取出有意义的特征。轮廓、直线、斑点等就是基本的图像图元，用来描述图像包含的元素。本章将介绍如何提取这些重要的图像特征。

7.2 用 Canny 算子检测图像轮廓

上一章我们学习了如何检测图像的边缘。尤其是通过对梯度幅值的阈值化，获得图像中主要边缘的二值分布图。边缘勾画出了图像的元素，含有重要的视觉信息。基于这个原因，边缘可应用于目标识别等领域。但是简单的二值边缘分布图有两个主要缺点。首先，检测到的边缘过厚，这导致更加难以识别物体的边界。第二，也是更重要的，通常不可能找到这样的阈值：低到足以检测到图像中所有重要的边缘，同时又高到足以避免产生太多无关紧要的边缘。这是一个难以权衡的问题，Canny算法试图解决这个问题。

7.2.1 如何实现

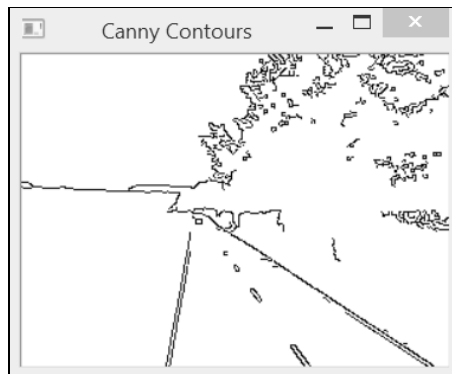
Canny算法可通过OpenCV的`cv::Canny`函数中实现。使用这个算法时需要指定两个阈值（后面会解释原因）。调用函数的方法如下：

```
// 应用Canny算法
cv::Mat contours;
cv::Canny(image,    // 灰度图像
          contours, // 输出轮廓
          125,      // 低阈值
          350);     // 高阈值
```

先来看这个图像：



在这个图像上应用Canny算法，得到结果如下：



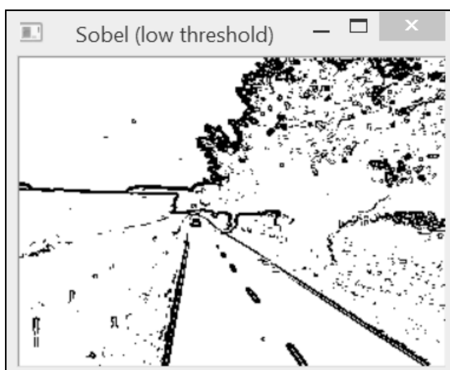
注意，由于正常的结果是用非零像素表示轮廓的，因此为了得到上面显示的图像，需要反转黑色和白色像素值。而上面显示的图像只是像素值为255的轮廓。

7.2.2　实现原理

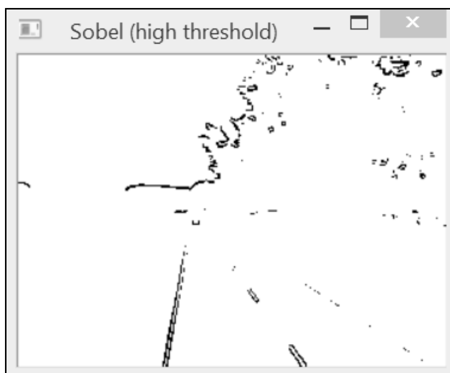
Canny算子通常基于第6章介绍的Sobel算子，虽然也可使用其他梯度算子。它的核心理念是用两个不同的阈值来判断哪个点属于轮廓：一个低阈值，一个高阈值。

选择低阈值时，要保证它能包含属于重要图像轮廓的全部边缘像素。例如，使用前面例子中

指定的低阈值，应用到Sobel算子返回的图像上，可得到如下边缘分布图：



可以看到，道路的边缘非常清晰。但因为这里使用了一个宽松的阈值，所以很多并不需要的边缘也被检测出来了。而第二个阈值的作用就是界定重要轮廓的边缘，它会排除掉异常的边缘。例如，在Sobel边缘分布图上应用上例中的高阈值后，得到结果如下：



现在我们得到的图像中有些边缘是断裂的，但是这些可见的边缘肯定属于本场景中重要的轮廓。Canny算法结合这两种边缘分布图，生成最优的轮廓分布图。具体做法是在低阈值边缘分布图上只保留具有连续路径的边缘点，同时把那些边缘点连接到属于高阈值边缘分布图的边缘上。这样一来，高阈值分布图上的所有边缘点都保留下来，而低阈值分布图上边缘点的孤立链全部被移除。这是一种很好的折中方案，只要指定适当的阈值，就能获得高质量的轮廓。这种基于两个阈值获得二值分布图的策略，称为*滞后阈值化*，可用于任何需要用阈值化获得二值分布图的场景。但是它的计算复杂度比较高。

另外，Canny算法用了一个额外的策略来优化边缘分布图的质量。在进行滞后阈值化之前，如果梯度幅值不是梯度方向上的最大值，那么对应的边缘点都会被移除。前面讲过，梯度的方向总是与边缘垂直的。因此这个方向上梯度的局部最大值，对应着轮廓最大强度的位置。这就是为

什么Canny轮廓分布图中边缘比较薄的原因。

7.2.3 参阅

- J.Canny的经典论文“A computational approach to edge detection”，发表于*IEEE Transactions on Pattern Analysis and Image Understanding*（1986年第6期第18卷）。

7.3 用霍夫变换检测直线

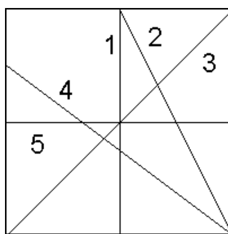
人造世界中充满了平面和线性结构。因此直线在图像中是很常见的。它们是很有意义的特征，在目标识别和图像理解领域有着非常重要的作用。霍夫变换（Hough）是一种常用于检测此类具体特征的经典算法。该算法起初用于检测图像中的直线，后来经过扩展，也能检测其他简单的图像结构。

7.3.1 准备工作

在霍夫变换中，用这个方程式表示直线：

$$\rho = x\cos\theta + y\sin\theta$$

参数ρ是直线与图像原点（左上角）的距离，θ是直线与垂直线的角度。在这种表示法中，图像中的直线有一个0到π（弧度）之间的角θ，而半径ρ的最大值是图像对角线的长度。例如，下面的一组线：



像直线1这样的垂直线，其角度值θ等于0，而水平线（例如直线5）的θ等于π/2。因此直线3的θ等于π/4，直线4大约是0.7π。为了表示[0, π]范围内的所有θ值，半径值可以用负数表示。例如直线2，它的θ等于0.8π，ρ是负数。

7.3.2 如何实现

针对用于检测直线的霍夫变换，OpenCV提供了两种实现方法。其中基础版是`cv::Hough-`

Lines。它输入的是一个二值分布图，包含了一批像素点（用非零像素表示），其中一些对齐的点构成直线。通常它是一个已经生成的边缘分布图，例如采用Canny算子生成的分布图。cv::HoughLines函数输出的是一个cv::Vec2f类型元素组成的向量，每个元素是一对浮点数，表示检测到的直线的参数，即(ρ, θ)。下面是使用这个函数的例子，首先用Canny算子获得图像轮廓，然后用霍夫变换检测直线：

```
// 应用Canny算法
cv::Mat contours;
cv::Canny(image,contours,125,350);
// 用霍夫变换检测直线
std::vector<cv::Vec2f> lines;
cv::HoughLines(test,lines,
     1,PI/180,  // 步长
     60);       // 最小投票数
```

第3个和第4个参数表示搜索直线时用的步长。在本例中，半径步长为1，表示函数将搜索所有可能的半径；角度步长为$\pi/180$，表示函数将搜索所有可能的角度。最后一个参数的功能将在下一段介绍。选用特定的参数后，可以从上一节的道路图像中检测到15条直线。为了让检测结果可视化，我们在原始图像上绘制这些直线。但是有一点需要强调，这个算法检测的是图像中的直线而不是线段，它不会给出直线的端点。因此我们绘制的直线将穿透整个图像。具体做法是，对于垂直方向的直线，我们计算它与图像水平边界（即第一行和最后一行）的交叉点，然后在这两个交叉点之间画线。对于水平方向的直线也类似，只不过用第一列和最后一列。画线的函数是cv::line。需要注意的是，即使点的坐标超出了图像范围，这个函数也能正确运行，因此没必要检查交叉点是否在图像内部。通过遍历直线向量画出所有直线，代码如下：

```
std::vector<cv::Vec2f>::const_iterator it= lines.begin();
while (it!=lines.end()) {

   float rho= (*it)[0]; // 第一个元素是距离rho
   float theta= (*it)[1]; // 第二个元素是角度theta

   if (theta < PI/4.
        || theta > 3.*PI/4.) { // 垂直线（大致）

      // 直线与第一行的交叉点
      cv::Point pt1(rho/cos(theta),0);
      // 直线与最后一行的交叉点
      cv::Point pt2((rho-result.rows*sin(theta))/
                               cos(theta),result.rows);
      // 画白色的线
      cv::line( image, pt1, pt2, cv::Scalar(255), 1);

   } else { // 水平线（大致）

      // 直线与第一列的交叉点
      cv::Point pt1(0,rho/sin(theta));
      // 直线与最后一列的交叉点
      cv::Point pt2(result.cols,
```

```
            (rho-result.cols*cos(theta))/sin(theta));
      // 画白色的线
      cv::line(image, pt1, pt2, cv::Scalar(255), 1);
   }

   ++it;
}
```

得到的结果如下：



可以看出，霍夫变换只是寻找图像中边缘像素的对齐区域。因为有些像素只是碰巧排成了直线，这种方式可能产生错误的检测结果。也可能因为多条直线穿过了同一个像素对齐区域，而检测出重复的结果。

为解决上述问题并检测到线段（即包含端点的直线），人们提出了霍夫变换的改进版。这就是概率霍夫变换，在OpenCV中通过`cv::HoughLinesP`函数实现。我们用它创建`LineFinder`类，封装函数的参数：

```
class LineFinder {

  private:

     // 原始图像
     cv::Mat img;

     // 包含被检测直线的端点的向量
     std::vector<cv::Vec4i> lines;

     // 累加器分辨率参数
     double deltaRho;
     double deltaTheta;

     // 确认直线之前必须收到的最小投票数
     int minVote;

     // 直线的最小长度
```

```
    double minLength;

    // 直线上允许的最大空隙
    double maxGap;

  public:

    // 默认累加器分辨率是1像素，1度
    // 没有空隙，没有最小长度
    LineFinder() : deltaRho(1), deltaTheta(PI/180),
                   minVote(10), minLength(0.), maxGap(0.) {}
```

看一下对应的设置方法：

```
// 设置累加器的分辨率
void setAccResolution(double dRho, double dTheta) {

   deltaRho= dRho;
   deltaTheta= dTheta;
}

// 设置最小投票数
void setMinVote(int minv) {

   minVote= minv;
}

// 设置直线长度和空隙
void setLineLengthAndGap(double length, double gap) {

   minLength= length;
   maxGap= gap;
}
```


用上述方法，检测霍夫线段的代码如下：

```
// 应用概率霍夫变换
std::vector<cv::Vec4i> findLines(cv::Mat& binary) {

   lines.clear();
   cv::HoughLinesP(binary,lines,
                   deltaRho, deltaTheta, minVote,
                   minLength, maxGap);

   return lines;
}
```

这个方法返回`cv::Vec4i`类型的向量，包含每条被检测线段的开始和结束端点的坐标。我们可以用下面的方法在图像上绘制检测到的线段：

```
// 在图像上绘制检测到的直线
void drawDetectedLines(cv::Mat &image,
          cv::Scalar color=cv::Scalar(255,255,255)) {
```

```
      // 画直线
      std::vector<cv::Vec4i>::const_iterator it2=
                                       lines.begin();

      while (it2!=lines.end()) {

         cv::Point pt1((*it2)[0],(*it2)[1]);
         cv::Point pt2((*it2)[2],(*it2)[3]);

         cv::line( image, pt1, pt2, color);

         ++it2;
      }
   }
```

现在用同一个输入图像，可以用下面的次序检测直线：

```
   // 创建LineFinder类的实例
   LineFinder finder;

   // 设置概率霍夫变换的参数
   finder.setLineLengthAndGap(100,20);
   finder.setMinVote(60);

   // 检测直线并画线
   std::vector<cv::Vec4i> lines= finder.findLines(contours);
   finder.drawDetectedLines(image);
   cv::namedWindow("Detected Lines with HoughP");
   cv::imshow("Detected Lines with HoughP",image);
```

上面的代码得到如下结果：



7.3.3　实现原理

霍夫变换的目的是在二值图像中找出全部直线，并且这些直线必须穿过足够多的像素点。它的处理方法是，检查输入的二值分布图中每个独立的像素点，识别出穿过该像素点的所有可能的

直线。如果同一条直线穿过很多像素点，就说明这条直线明显到足以被认定。

为了统计某条直线被标识的次数，霍夫变换使用一个二维累加器。累加器的大小依据(ρ, θ)的步长确定，其中(ρ, θ)参数用来表示一条直线。为了说明霍夫变换的功能，我们建立一个180×200的矩阵（对应θ的步长为π/180，ρ的步长为1）：

```
// 创建霍夫累加器
// 这里的图像类型为uchar；实际使用时应该用int
cv::Mat acc(200,180,CV_8U,cv::Scalar(0));
```

累加器是不同于(ρ, θ)值的映射表。因此，矩阵的每个入口都对应一条特定的直线。现在我们假定某个像素点的坐标为(50, 30)，这样就可以循环遍历所有可能的θ值（步长π/180），并计算对应的（四舍五入）ρ值，从而标识出穿过这个像素点的全部直线：

```
// 选取一个像素点
int x=50, y=30;

// 循环遍历所有角度
for (int i=0; i<180; i++) {

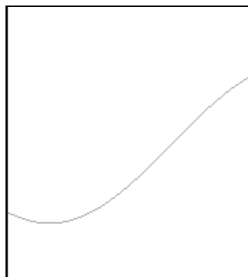
   double theta= i*PI/180.;

   // 找到对应的rho值
   double rho= x*std::cos(theta)+y*std::sin(theta);
   // j对应在-100到100范围内的rho
   int j= static_cast<int>(rho+100.5);

   std::cout << i << "," << j << std::endl;

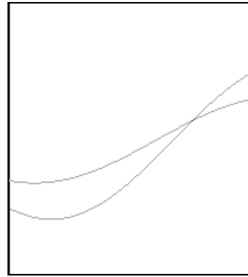
   // 增值累加器
   acc.at<uchar>(j,i)++;
}
```

每次计算得到(ρ, θ)对后，其对应的累加器入口的数值就会增加，表示对应的直线穿过了图像中的某个像素点（或者说，每个像素点为一批候选直线投票）。如果把累加器作为图像显示（翻转过来，并乘以100，以便数字1能显示），结果如下：



上面的曲线表示穿过这个点的所有直线的集合。现在我们用像素点(30, 10)重复上述过程，得

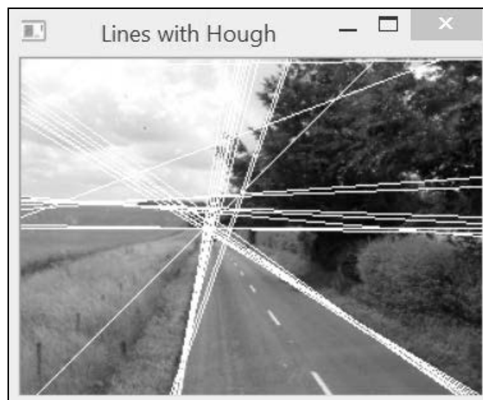
到的累加器如下所示：



可以看到，这两条曲线在一个位置相交：这个位置表示对应的直线通过了这两个像素点。累加器的对应入口收到了两次投票，表明有两个像素点在这条直线上。如果对二值分布图中的所有像素点重复上述过程，那么同一条直线上的像素点会使累加器的同一个入口增长很多次。最后，为了检测图像中的直线（即像素点对齐的位置），只需要标识出累加器中的局部限值，该累加器用于接收大量投票数。cv::HoughLines函数的最后一个参数就表示最低投票数，只有不低于这个数的直线才会被检测到。例如我们把这个数字降低到50，如下：

```
cv::HoughLines(test,lines,1,PI/180,50);
```

用这个代码，就会在前面的图像中检测到更多的直线，如下图所示：



概率霍夫变换对基本算法做了一些修正。首先，概率霍夫变换在二值分布图上随机选择像素点，而不是系统性地逐行扫描图像。一旦累加器的某个入口达到了预设的最小值，就沿着对应的直线扫描图像，并移除所有在这条直线上的像素点（包括还没投票的像素点）。这个扫描过程还检测可以接受的线段长度。为此，算法定义了两个额外的参数：一个是允许的线段最小长度，另一个是组成连续线段时允许的最大像素间距。这个额外的步骤增加了算法的复杂度，但也得到了一定的补偿，即由于在扫描直线过程中已经清除了部分像素点，因此减少了投票过程中用到的像素点。

7.3.4 扩展阅读

霍夫变换也能用来检测其他几何物体。事实上，任何可以用一个参数方程来表示的物体，都很适合用霍夫变换来检测。

检测圆

圆的参数方程为：

$$r^2 = (x - x_0)^2 + (y - y_0)^2$$

这个方程包含三个参数（圆半径和圆心坐标），这表明需要使用三维的累加器。然而，通常情况下，累加器的维数越多，霍夫变换的可靠性就越低。在本例中，每个像素点都会使累加器增加大量的入口。因此，精确地定位局部尖峰值会变得更加困难。为解决这个问题，人们提出了各种策略。OpenCV采用的策略是在用霍夫变换检测圆的实现中使用两轮筛选。第一轮筛选使用一个二维累加器，找出可能是圆的位置。因为圆周上像素点的梯度方向与半径的方向是一致的，对于每个像素点，累加器中只对沿着梯度方向的入口增加计数（根据预先定义的最小和最大半径值）。一旦检测到可能的圆心（即收到了预定数量的投票），就在第二轮筛选中建立半径值范围的一维直方图。这个直方图的尖峰值就是被检测圆的半径。

实现上述策略的`cv::HoughCircles`函数结合了Canny检测和霍夫变换。它的调用方法是：

```
cv::GaussianBlur(image,image,cv::Size(5,5),1.5);
std::vector<cv::Vec3f> circles;
cv::HoughCircles(image, circles, CV_HOUGH_GRADIENT,
   2,   // 累加器分辨率（图像尺寸/2）
   50,  // 两个圆之间的最小距离
   200, // Canny算子的高阈值
   100, // 最少投票数
   25, 100); // 最小和最大半径
```

有一点需要反复提醒，就是在调用`cv::HoughCircles`函数之前要对图像进行平滑化，以减少图像中可能导致误判的噪声。检测的结果存放在`cv::Vec3f`实例的向量中。前面两个数值是圆心坐标，第三个数值是半径。

编写本书时，`CV_HOUGH_GRADIENT`是唯一可用的参数，它代表两轮筛选的圆形检测方法。第四个参数定义了累加器的分辨率，它是一个分割比例。例如，数值2表示累加器是图像尺寸的一半。下一个参数是两个被检测的圆之间的最小像素距离。另一个参数是Canny边缘检测器的高阈值。低阈值通常设置为高阈值的一半。第七个参数是圆心位置必须收到的最少投票数，只有在第一轮筛选时收到的投票数超过该值，才能作为候选的圆进入第二轮筛选。最后的两个参数是被检测圆的最小和最大半径值。可以看出，这个函数包含的参数太多了，很难调节。

得到存放圆的向量后，就可以在图像上画出这些圆。方法是迭代遍历该向量，并调用

cv::circle函数，传入获得的参数：

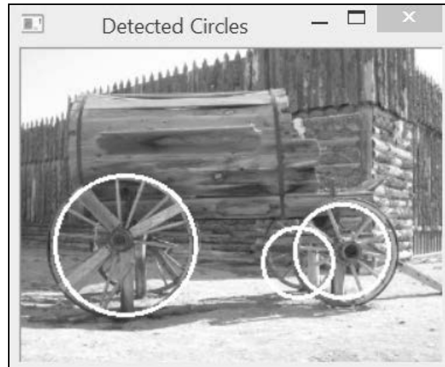
```
std::vector<cv::Vec3f>::
       const_iterator itc= circles.begin();

while (itc!=circles.end()) {

  cv::circle(image,
     cv::Point((*itc)[0], (*itc)[1]), // 圆心
     (*itc)[2],        // 半径
     cv::Scalar(255), // 颜色
     2);               // 厚度

  ++itc;
}
```

使用上述方法和参数在测试图像上执行，得到如下结果：



7.3.5　参阅

- C. Galambos、J. Kittler和J. Matas发表在*IEE Vision Image and Signal Processing* 2002年第148卷第3期第158页至第165页上的文章“Gradient-based Progressive Probabilistic Hough Transform”对霍夫变换的方法进行了大量引用，并且描述了OpenCV中实现的概率算法。
- H.K. Yuen、J. Princen、J. Illingworth和J Kittler发表在*Image and Vision Computing* 1990年第8卷第1期第71页至第77页上的文章“Comparative Study of Hough Transform Methods for Circle Finding”描述了用霍夫变换检测圆的各种策略。

7.4　点集的直线拟合

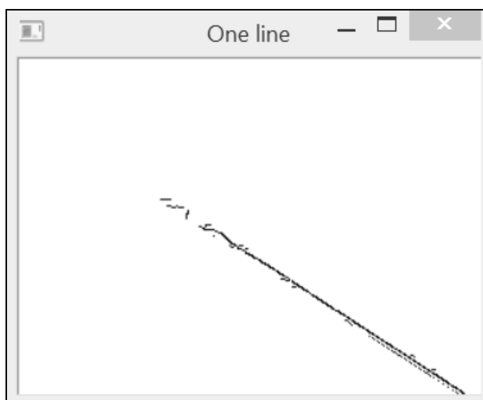
在某些应用中，重要的不仅是检测出图像中的直线，还需要精确地估计直线的位置和方向。本节介绍如何找到最适合指定点集的直线。

7.4.1 如何实现

首先需要识别出图像中靠近直线的点。我们使用一条上节检测到的直线。把cv::HoughLinesP检测到的直线存放在std::vector<cv::Vec4i>类型的变量lines中。为了提取出靠近第一条直线的点集，可以继续以下步骤。在黑色图像上画一条白色直线，并且穿过用于检测直线的Canny轮廓图。这可以用这些语句实现：

```
int n=0; // 选用直线0
// 黑色图像
cv::Mat oneline(contours.size(),CV_8U,cv::Scalar(0));
// 白色直线
cv::line(oneline,
         cv::Point(lines[n][0],lines[n][1]),
         cv::Point(lines[n][2],lines[n][3]),
         cv::Scalar(255),
         3); // 直线宽度
// 轮廓与白色直线进行“与”运算
cv::bitwise_and(contours,oneline,oneline);
```

结果是一个只包含了与指定直线相关的点的图像。为了引入公差，我们画了具有一定宽度（这里是3）的直线。因此位于指定邻域内的点都被接受。下面是获得的图像（为了显示效果更好，对其做了反转）：



然后可以把这些点的坐标插入到cv::Point对象的std::vector类型中（也可以使用浮点数坐标，即cv::Point2f），代码如下：

```
std::vector<cv::Point> points;

// 迭代遍历像素，得到所有点的位置
for( int y = 0; y < oneline.rows; y++ ) {
   // 行y

   uchar* rowPtr = oneline.ptr<uchar>(y);
```

```
    for( int x = 0; x < oneline.cols; x++ ) {
       // 列x

       // 如果在轮廓上
       if (rowPtr[x]) {

          points.push_back(cv::Point(x,y));
       }
    }
}
```

用OpenCV的函数`cv::fitLine`，可以很容易地得到最优的拟合直线：

```
cv::Vec4f line;
cv::fitLine(points,line,
            CV_DIST_L2, // 距离类型
            0,          // L2距离不用这个参数
            0.01,0.01); // 精度
```

上述代码用直线方程式作为参数，它的形式是一个单位方向向量（`cvVec4f`的前两个数值）和直线上一个点的坐标（`cvVec4f`的后两个数值）。本例中，方向向量是(0.83, 0.55)，点的坐标是(366.1, 289.1)。最后两个参数是所需的直线精度。

直线方程式通常用于某些属性的计算（例如需要精确参数的校准）。为了演示它的用法，也为了验证计算的直线是否正确，我们在图像上模拟一条直线。这里只是随意地画一条长度为100像素，宽度为3像素的黑色线段：

```
int x0= line[2];         // 直线上的一个点
int y0= line[3];
int x1= x0+100*line[0]; // 加上长度为100的向量（用单位向量生成）
int y1= y0+100*line[1];
// 绘制这条线
cv::line(image,cv::Point(x0,y0),cv::Point(x1,y1),
         0,3); // 颜色和宽度
```

结果如下：



7.4.2　实现原理

点集的直线拟合是一个经典的数学问题。OpenCV的实现方法，是使每个点到直线的距离之和最小化。在众多用于计算距离的函数中，欧几里德距离的计算速度最快，所用参数为CV_DIST_L2。这一选项对应了标准的最小二乘法直线拟合。如果点集中包含了孤立点（即不属于直线的点），可以选用其他距离函数，以减少远距离的点带来的影响。最小化计算的基础是M估算法技术，该算法采用迭代方式解决加权最小二乘法问题，其中权重与点到直线的距离成反比。

我们也可以用这个函数在三维点集上拟合直线。这时输入的是cv::Point3i或cv::Point3f对象的集合，输出的是一个std::Vec6f实例。

7.4.3　扩展阅读

cv::fitEllipse函数在二维点集上拟合一个椭圆。它返回一个旋转的矩形（一个cv::RotatedRect实例），矩形中有一个内切的椭圆。对应的代码如下：

```
cv::RotatedRect rrect= cv::fitEllipse(cv::Mat(points));
cv::ellipse(image,rrect,cv::Scalar(0));
```

7.5　提取区域的轮廓

图像通常包含各种物体，图像分析的目的之一就是识别和提取这些物体。在物体检测和识别程序中，第一步通常就是生成二值图像，以得到被关注物体所处的位置。不管用什么方式获得二值图像（例如用第4章的直方图反向投影，或者用第11章的运动分析），下一个步骤都是从由1和0组成的像素集合中提取出物体。

例如我们来看第5章的含有水牛的二值图像，如下图所示：



获得这个图像的方法是执行一次简单的阈值化操作，然后应用开启和闭合形态学滤波器。本节将介绍如何从这样的图像中提取物体。具体来说，就是提取连通区域，即二值图像中由一批连

通的像素构成的形状。

7.5.1　如何实现

OpenCV提供了一个简单的函数，可以提取出图像中连通区域的轮廓。这个函数就是cv::findContours：

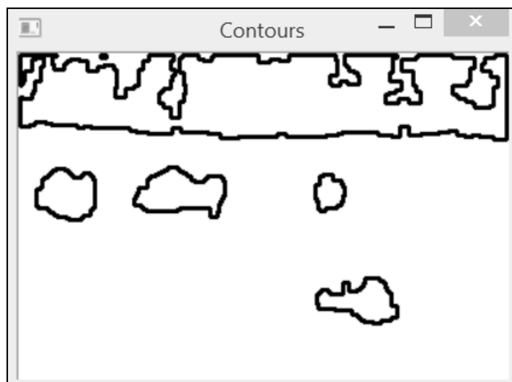
```
// 用于存储轮廓的向量
std::vector<std::vector<cv::Point>> contours;
cv::findContours(image,
   contours, // 存储轮廓的向量
   CV_RETR_EXTERNAL, // 检索外部轮廓
   CV_CHAIN_APPROX_NONE); // 每个轮廓的全部像素
```

显然，函数输入的就是上述二值图像。输出的是一个存储轮廓的向量，每个轮廓用一个cv::Point类型的向量表示。因此输出参数是一个由std::vector实例构成的std::vector实例。并且指明了两个选项。第一个选项表示只检索外部轮廓，即物体内部的空穴会被忽略（7.5.3节将讨论其他的选项）。第二个选项指明了轮廓的格式。使用当前的选项，向量将列出轮廓的全部点。如使用CV_CHAIN_APPROX_SIMPLE，则只会列出包含水平、垂直或对角线轮廓的端点。用其他选项可得到逼近轮廓的更复杂的链，对轮廓的表示将更紧凑。在前面的图像中可检测到9个连通区域，用contours.szie()查看轮廓的数量。

有一个非常实用的函数，可在图像（这里用白色图像）上画出那些区域的轮廓：

```
// 在白色图像上画黑色轮廓
cv::Mat result(image.size(),CV_8U,cv::Scalar(255));
cv::drawContours(result,contours,
   -1, // 画全部轮廓
    0, // 用黑色画
    2);// 宽度为2
```

如果这个函数的第三个参数是负数，就画出全部轮廓，否则就可以指定要画的轮廓的序号。得到的结果如下：



7.5.2 实现原理

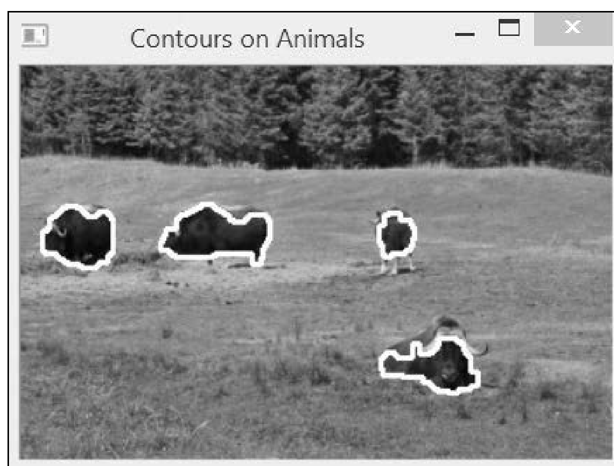
提取轮廓的算法很简单，它系统地扫描图像，直到找到连通区域。从区域的起点开始，沿着它的轮廓对边界像素做标记。处理完这个轮廓后就从上个位置恢复扫描过程，直到发现新的区域。

然后可以对识别出的连通区域进行独立的分析。例如，如果事先已经知道所关注物体的大小，就可以将部分区域删除。我们采用区域边界的最小和最大值。具体做法是迭代遍历存放轮廓的向量，并且删除无效的轮廓：

```
// 删除太短或太长的轮廓
int cmin= 50; // 最小轮廓长度
int cmax= 1000; // 最大轮廓长度
std::vector<std::vector<cv::Point>>::
          iterator itc= contours.begin();
// 针对所有轮廓
while (itc!=contours.end()) {

   // 验证轮廓大小
   if (itc->size() < cmin || itc->size() > cmax)
      itc= contours.erase(itc);
   else
      ++itc;
}
```

因为在`std::vector`中的删除操作的时间复杂度为O(N)，所以这个循环的效率还可以更高。不过对于小型向量，总体的开销并不大。这次我们在原始图像上画出剩下的轮廓，结果如下：



这个图像刚好有这种简单的规则，可用来识别所有的关注目标。在更复杂的情况下，就需要对区域的属性做更精细的分析。这正是下一节要做的。

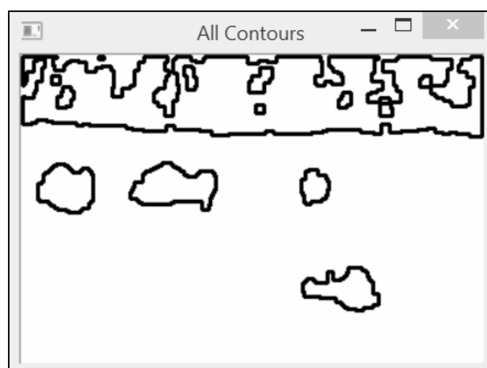
7

7.5.3　扩展阅读

cv::findContours函数也能检测二值图像中所有的闭合轮廓，包括区域内部空穴构成的轮廓。实现方法是在调用函数时指定另一个标志：

```
cv::findContours(image,
   contours, // 存放轮廓的向量
   CV_RETR_LIST, // 检索全部轮廓
   CV_CHAIN_APPROX_NONE); // 每个轮廓的全部像素
```

调用后得到如下轮廓：



注意，背景森林中增加了额外的轮廓。也可以把这些轮廓分层次组织起来。主区域是父轮廓，它内部的空穴是它的子轮廓，如果空穴内部还有区域，那它们就是上述子轮廓的子轮廓，以此类推。使用CV_RETR_TREE标志可得到这个层次结构，代码为：

```
std::vector<cv::Vec4i> hierarchy;
cv::findContours(image,
   contours, // 存放轮廓的向量
   hierarchy, // 层次结构
   CV_RETR_TREE, // 用树状结构式检索全部轮廓
   CV_CHAIN_APPROX_NONE); // 每个轮廓的全部像素
```

本例中每个轮廓都有一个对应的层次元素，存放次序与轮廓相同。层次元素由四个整数构成，前两个整数是下一个和上一个同级轮廓的序号，后两个整数是第一个子轮廓和父轮廓的序号。如果序号为负数，就表示轮廓列表的末端。CV_RETR_CCOMP标志的作用与之类似，但只允许两个层次。

7.6　计算区域的形状描述子

连通区域通常代表着场景中的某个物体。为了识别该物体，或将它与其他图像元素比较，需要对此区域进行测量，以提取出部分特征。本节介绍几种OpenCV的形状描述子，用于描述连通区域的形状。

7.6.1 如何实现

OpenCV中用于形状描述的函数有很多。我们把其中的几个函数应用到上节提取的区域。特别是使用包含四个轮廓的向量，这些轮廓分别代表前面已经识别的四头水牛。在下面的代码段中，我们将计算轮廓的形状描述子（从contours[0]到contours[3]），并在轮廓图像（宽度为1）上画出结果（宽度为2）。图像见本段后面。

第一个是边界框，用于右下角的区域：

```
// 测试边界框
cv::Rect r0= cv::boundingRect(contours[0]);
// 画矩形
cv::rectangle(result,r0, 0, 2)
```

最小覆盖圆的情况也类似。它用于右上角的区域：

```
// 测试覆盖圆
float radius;
cv::Point2f center;
cv::minEnclosingCircle(contours[1],center,radius);
// 画圆形
cv::circle(result,center,
          static_cast<int>(radius),cv::Scalar(0),2);
```

计算区域轮廓的多边形逼近的代码如下（位于左侧区域）：

```
// 测试多边形逼近
std::vector<cv::Point> poly;
cv::approxPolyDP(contours[2],poly,5,true);
// 画多边形
cv::polylines(result, poly, true, 0, 2);
```

7

注意多边形绘制函数cv::polylines，它与其他画图函数很相似。第三个布尔型参数表示轮该廓是否闭合（如果闭合，最后一个点将与第一个点相连）。

凸包是另一种形式的多边形逼近（位于左边第二个区域）：

```
// 测试凸包
std::vector<cv::Point> hull;
cv::convexHull(contours[3],hull);
// 画多边形
cv::polylines(result, hull, true, 0, 2);
```

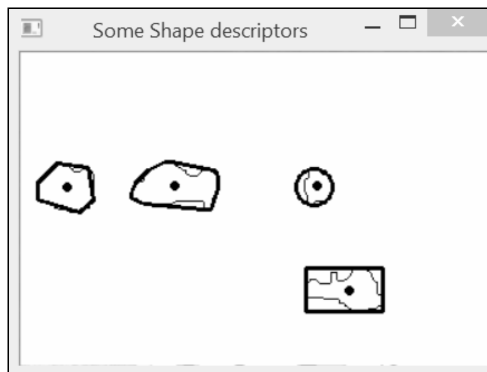
最后，计算轮廓矩是另一种功能强大的描述子（在所有区域内部画出重心）：

```
// 测试轮廓矩
// 迭代遍历所有轮廓
itc= contours.begin();
while (itc!=contours.end()) {
```

```
    // 计算全部轮廓矩
    cv::Moments mom= cv::moments(cv::Mat(*itc++));

    // 画重心
    cv::circle(result,
       // 重心位置转换成整数
       cv::Point(mom.m10/mom.m00,mom.m01/mom.m00),
       2,cv::Scalar(0),2); // 画黑点
}
```

结果如下：



7.6.2　实现原理

在表示和定位图像中的区域方法中，边界框可能是最简洁的。它的定义是：能完整包含该形状的最小垂直矩形。比较边界框的高度和宽度，可以获得物体的在垂直或水平方向的范围。最小覆盖圆通常用在只需要区域尺寸和位置的近似值的情况。

如果要更紧凑地表示区域的形状，可采用多边形逼近。在创建时要制定精确度参数，表示形状与对应的简化多边形之间能接受的最大距离。它是cv::approxPolyDP函数的第四个参数。返回的结果是cv::Point类型的向量，表示多边形的顶点个数。在画这个多边形时，要迭代遍历整个向量，并在顶点之间画直线，把它们逐个连接起来。

形状的凸包（或凸包络），是包含该形状的最小凸多边形。可以把它看作一条绕在区域周围的橡皮筋。可以看出，在形状轮廓中凹进去的位置，凸包轮廓会与原始轮廓发生偏离。

通常可用凸包缺陷来表示这些位置，OpenCV中有一个专门用于识别凸包缺陷的函数：cv::convexityDefects。调用方法如下：

```
std::vector<cv::Vec4i> defects;
cv::convexityDefects(contour, hull, defects);
```

参数contour和hull分别表示原始轮廓和凸包轮廓（两者都用std::vector<cv::Point>

的实例表示）。函数输出的是一个向量，它的每个元素由四个整数组成。前两个整数是顶点在轮廓中的索引，用来界定该缺陷；第三个整数表示凹陷内部最远的点；最后的整数表示最远点与凸包之间的距离。

轮廓矩是形状结构分析中常用的数学模型。OpenCV定义了一个数据结构，封装了形状中计算得到的所有轮廓矩。它是函数`cv::moments`的返回值。这些轮廓矩共同代表了物体形状的紧凑程度，常用于特征识别。我们只是用该结构获得每个区域的重心，这里用前面三个空间轮廓矩计算得到。

7.6.3 扩展阅读

可以用OpenCV函数计算得到其他结构化属性。函数`cv::minAreaRect`计算最小覆盖自由矩形（在5.6节用到了这个函数）。函数`cv::contourArea`估算轮廓的面积（内部的像素数量）。函数`cv::pointPolygonTest`判断一个点在轮廓的内部还是外部；函数`cv::matchShapes`度量两个轮廓之间的相似度。所有这些度量属性的方法都可以有效地结合起来，用于更高级的结构分析。

四边形检测

第5章讲到的MSER特征，可作为一种从图像提取形状的高效工具。利用前面章节中用MSER得到的结果，现在我们来构建一个在图像中监测四边形区域的算法。在当前图像中，该算法可用于检测建筑物上的窗户。可以很容易地获得MSER的二值图像：

```
// 创建二值图像
components= components==255;
// 打开图像（包含背景）
cv::morphologyEx(components,components,
                 cv::MORPH_OPEN,cv::Mat(),
                 cv::Point(-1,-1),3);
```

另外，我们用形态学滤波器来清理图像。得到的图像如下：



7

下一步是获取轮廓：

```
//翻转图像（背景必须是黑色的）
cv::Mat componentsInv= 255-components;
// 得到连通区域的轮廓
cv::findContours(componentsInv,
  contours, // 轮廓的向量
  CV_RETR_EXTERNAL, // 检索外部轮廓
  CV_CHAIN_APPROX_NONE);
```

最后得到全部轮廓，并用多边形粗略地逼近它们：

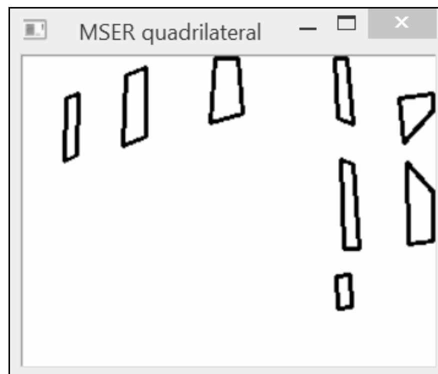
```
// 白色图像
cv::Mat quadri(components.size(),CV_8U,255);

// 针对全部轮廓
std::vector<std::vector<cv::Point>>::iterator
                            it= contours.begin();
while (it!= contours.end()) {
  poly.clear();
  // 用多边形逼近轮廓
  cv::approxPolyDP(*it,poly,10,true);

  // 是否四边形?
  if (poly.size()==4) {

        // 画出来
    cv::polylines(quadri, poly, true, 0, 2);
  }
  ++it;
}
```

四边形就是有四条边的多边形。检测结果如下：



要检测矩形，只需测量相邻边的夹角，并且排除夹角与90度相差很大的四边形。

第 8 章

检测兴趣点

8

本章包括以下内容：

- ❑ 检测图像中的角点；
- ❑ 快速检测特征；
- ❑ 尺度不变特征的检测；
- ❑ 多尺度FAST特征的检测。

8.1 简介

在计算机视觉领域，兴趣点（也称关键点或特征点）的概念已经得到了广泛的应用，包括目标识别、图像配准、视觉跟踪、三维重建等。这个概念的原理是，从图像中选取某些特征点并对图像进行局部分析，而非观察整幅图像。只要图像中有足够多可检测的兴趣点，并且这些兴趣点各不相同且特征稳定，能被精确地定位，上述方法就十分有效。

因为要用于图像内容的分析，所以不管图像拍摄时采用了什么视角、尺度和方位，理想情况下同一个场景或目标位置都要检测到特征点。视觉不变性是图像分析中一个非常重要的属性，目前有大量关于它的研究。我们将会看到，各种检测方法具有不同的不变性。本章重点关注的是关键点提取这一过程本身。后面两章将介绍兴趣点如何应用在各个方面，例如图像匹配或图像几何估计。

8.2 检测图像中的角点

在图像中搜索有价值的特征点时，使用角点是一种不错的方法。角点是很容易在图像中定位的局部特征，并且大量存在于人造物体中（例如墙壁、门、窗户、桌子等产生的角点）。角点的价值在于它是两条边缘线的接合点，是一种二维特征，可以被精确地定位（即使是子像素级精度）。与此相反的是位于均匀区域或物体轮廓上的点以及在同一物体的不同图像上很难重复精确定位的点。Harris特征检测是检测角点的经典方法。本节将详细探讨这个方法。

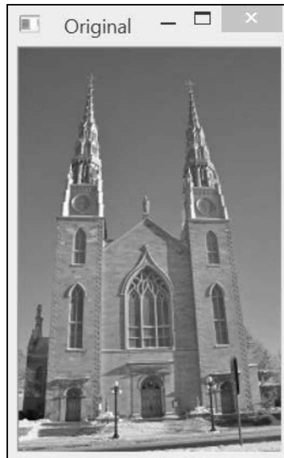
8.2.1　如何实现

OpenCV中检测Harris角点的基本函数是cv::cornerHarris，它非常易于使用。调用该函数时输入一个图像，返回的结果是一个浮点数型图像，其中每个像素表示角点强度。然后对输出图像阈值化，以获得检测角点的集合。代码如下：

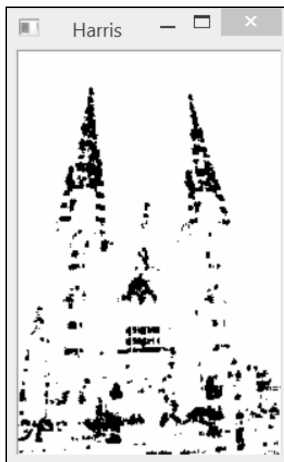
```
// 检测Harris角点
cv::Mat cornerStrength;
cv::cornerHarris(image,       // 输入图像
             cornerStrength, // 角点强度的图像
             3,              // 邻域尺寸
             3,              // 口径尺寸
             0.01);          // Harris参数

// 对角点强度阈值化
cv::Mat harrisCorners;
double threshold= 0.0001;
cv::threshold(cornerStrength,harrisCorners,
              threshold,255,cv::THRESH_BINARY);
```

这是原始图像：



结果是一个二值分布图像，如下图所示。这种反转处理是为了更直观地观察结果（即用cv::THRESH_BINARY_INV代替cv::THRESH_BINARY，用黑色表示被检测的角点）：



在前面的函数中，我们发现兴趣点检测法需要使用几个参数（下一节会详细解释），这可能会导致该方法很难调节。而且得到的角点分布图中包含很多聚集的角点像素，而不是我们想要检测出的具有明确定位的角点。因此我们定义一个检测Harris角点的类，以改进角点检测方法。

这个类封装了带有缺省值的Harris参数，以及对应的获取方法和设置方法（这里没有列出）：

```
class HarrisDetector {

  private:

     // 32位浮点数型的角点强度图像
     cv::Mat cornerStrength;
     // 32位浮点数型的阈值化角点图像
     cv::Mat cornerTh;
     // 局部最大值图像（内部）
     cv::Mat localMax;
     // 平滑导数的邻域尺寸
     int neighbourhood;
     // 梯度计算的口径
     int aperture;
     // Harris参数
     double k;
     // 阈值计算的最大强度
     double maxStrength;
     // 计算得到的阈值（内部）
     double threshold;
     // 非最大值抑制的邻域尺寸
     int nonMaxSize;
     // 非最大值抑制的内核
     cv::Mat kernel;

  public:

     HarrisDetector() : neighbourhood(3), aperture(3),
```

```
                        k(0.01), maxStrength(0.0),
                        threshold(0.01), nonMaxSize(3) {

        // 创建用于非最大值抑制的内核
        setLocalMaxWindowSize(nonMaxSize);
    }
```

检测Harris角点需要两个步骤。首先是计算每个像素的Harris值：

```
// 计算Harris角点
void detect(const cv::Mat& image) {

   // 计算Harris
   cv::cornerHarris(image,cornerStrength,
           neighbourhood,// 邻域尺寸
           aperture,     // 口径尺寸
           k);           // Harris参数

   // 计算内部阈值
   cv::minMaxLoc(cornerStrength,
        0, maxStrength);

   // 检测局部最大值
   cv::Mat dilated; // 临时图像
   cv::dilate(cornerStrength,dilated,cv::Mat());
   cv::compare(cornerStrength,dilated,
               localMax,cv::CMP_EQ);
}
```

然后是用指定的阈值获得特征点。因为Harris值的可选范围取决于选择的参数，所以阈值被作为质量等级，用最大Harris值的一个比例值表示：

```
// 用Harris值得到角点分布图
cv::Mat getCornerMap(double qualityLevel) {

   cv::Mat cornerMap;

   // 对角点强度阈值化
   threshold= qualityLevel*maxStrength;
   cv::threshold(cornerStrength,cornerTh,
                 threshold,255,cv::THRESH_BINARY);

   // 转换成8位图像
   cornerTh.convertTo(cornerMap,CV_8U);

   // 非最大值抑制
   cv::bitwise_and(cornerMap,localMax,cornerMap);

   return cornerMap;
}
```

这个方法返回一个被检测特征的二值角点分布图。因为Harris特征的检测过程分为两个方法，所以我们可以用不同的阈值来测试检测结果（直到获得适当数量的特征点），而不必重复耗时的

计算过程。也可以从`cv::Point`类型的`std::vector`中得到Harris特征：

```
// 用Harris值得到角点分布图
void getCorners(std::vector<cv::Point> &points,
                double qualityLevel) {

   // 获得角点分布图
   cv::Mat cornerMap= getCornerMap(qualityLevel);
   // 获得角点
   getCorners(points, cornerMap);
}

// 用角点分布图得到特征点
void getCorners(std::vector<cv::Point> &points,
                const cv::Mat& cornerMap) {

   // 迭代遍历像素，得到所有特征
   for( int y = 0; y < cornerMap.rows; y++ ) {

      const uchar* rowPtr = cornerMap.ptr<uchar>(y);

      for( int x = 0; x < cornerMap.cols; x++ ) {

         // 如果它是一个特征点
         if (rowPtr[x]) {

            points.push_back(cv::Point(x,y));
         }
      }
   }
}
```

这个类通过增加非最大值抑制步骤，也改进了Harris角点检测过程，下一节会详细解释这个步骤。现在可以用`cv::circle`函数画出检测到的特征点，方法如下：

```
// 在特征点的位置画圆形
void drawOnImage(cv::Mat &image,
   const std::vector<cv::Point> &points,
   cv::Scalar color= cv::Scalar(255,255,255),
   int radius=3, int thickness=1) {

   std::vector<cv::Point>::const_iterator it=
                               points.begin();

   // 针对所有角点
   while (it!=points.end()) {

      // 在每个角点位置画一个圆
      cv::circle(image,*it,radius,color,thickness);
      ++it;
   }
}
```

使用这个类检测Harris特征点的方法如下：

```
// 创建Harris检测器实例
HarrisDetector harris;
// 计算Harris值
harris.detect(image);
// 检测Harris角点
std::vector<cv::Point> pts;
harris.getCorners(pts,0.02);
// 画出Harris角点
harris.drawOnImage(image,pts);
```

结果如下图所示：



8.2.2　实现原理

为了定义图像中角点的概念，Harris特征检测方法在假定的兴趣点周围放置一个小窗口，并观察窗口内某个方向上强度值的平均变化。如果位移向量为$(\boldsymbol{u}, \boldsymbol{v})$，那么平均强度值变化就是：

$$\boldsymbol{R} \approx \sum (I(x+\boldsymbol{u}, y+\boldsymbol{v}) - I(x,y))^2$$

累加的范围是该像素周围一个预先定义的邻域（邻域的尺寸取决于`cv::cornerHarris`函数的第三个参数）。在所有方向上计算平均强度变化值，如果多个方向的变化值都很高，就认为这个点是角点。根据这个定义，Harris测试的步骤如下。首先获得平均强度值变化最大的方向。然后检查垂直方向上的平均强度变化值，看它是否也很大。如果是，就说明这是一个角点。

在数学上，可以用泰勒展开式近似地计算上述公式，验证这个判断：

$$\boldsymbol{R} \approx \sum \left((I(x+y) + \frac{\partial I}{\partial x}\boldsymbol{u} + \frac{\partial I}{\partial y}\boldsymbol{v} - I(x,y) \right)^2 = \sum \left(\left(\frac{\partial I}{\partial x}\boldsymbol{u} \right)^2 + \left(\frac{\partial I}{\partial y}\boldsymbol{v} \right)^2 + 2\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\boldsymbol{uv} \right)$$

写成矩阵形式，就是：

$$R \approx [\boldsymbol{u} \quad \boldsymbol{v}] \begin{bmatrix} \sum\left(\frac{\delta I}{\delta x}\right)^2 & \sum\frac{\delta I}{\delta x}\frac{\delta I}{\delta y} \\ \sum\frac{\delta I}{\delta x}\frac{\delta I}{\delta y} & \sum\left(\frac{\delta I}{\delta x}\right)^2 \end{bmatrix} \begin{bmatrix} \boldsymbol{u} \\ \boldsymbol{v} \end{bmatrix}$$

这是一个协方差矩阵，表示在所有方向上强度值变化的速率。这个定义包括了图像的一阶导数，通常用Sobel算子计算一阶导数。在OpenCV的实现方式中，这是函数的第四个参数，表示计算Sobel滤波器时用的口径。这个协方差矩阵的两个特征值，分别表示最大平均强度值变化和垂直方向的平均强度值变化。如果这两个特征值都很小，就说明是在相对同质的区域。如果一个特征值很大，另一个很小，那肯定是在边缘上。最后，如果两者都很大，那么就是在角点的位置。因此，判断一个点为角点的条件是它的协方差矩阵的最小特征值必须大于指定的阈值。

Harris角点算法的原始定义用到了特征分解理论的一些属性，以避免显式地计算特征值带来的开销。这些属性是：

- 矩阵的特征值之积等于它的行列式值；
- 矩阵的特征值之和等于它的对角元素之和（也就是矩阵的迹）。

我们可以通过计算下面的评分，来验证矩阵的特征值高不高：

$$Det(C) - kTrace^2(C)$$

只要两个特征值都高，就很容易证明这个评分肯定也高。这个评分由函数`cv::cornerHarris`在每个像素的位置计算得到。数值k是函数的第五个参数。确定这个参数的最佳值是比较困难的。但是根据经验，它在0.05和0.5之间通常能得到较 好的结果。

为了提升检测的效果，前面介绍的类增加了一个额外的非最大值抑制步骤。它的作用是排除掉紧邻的Harris角点。因此要被认定为是Harris角点，不仅要有高于指定阈值的评分，还必须是局部范围内的最大值。为了检查这个条件，`detect`方法中加入了一个小技巧，即对Harris评分的图像做膨胀运算：

```
cv::dilate(cornerStrength,dilated,cv::Mat());
```

膨胀运算会在邻域中把每个像素值替换成最大值，因此只有局部最大值的像素是不变的。用下面的相等测试可以验证这一点：

```
cv::compare(cornerStrength,dilated,
            localMax,cv::CMP_EQ);
```

因此矩阵`localMax`只有在局部最大值的位置才为真（即非零）。然后可在`getCornerMap`方法中，用它排除掉所有非最大值的特征（用`cv::bitwise_and`函数）。

8.2.3　扩展阅读

还可以对原始Harris角点检测算法做进一步的优化。本段介绍OpenCV的另一种角点检测方法，它扩展了Harris检测法，使角点在图像中的分布更加均匀。我们将看到，在OpenCV2通用接口中实现了该方法。

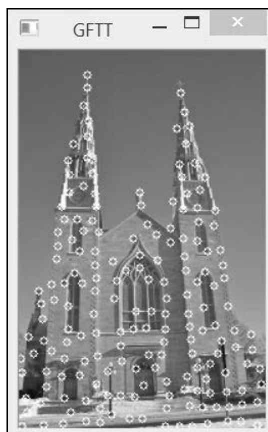
1. 适合跟踪的特征

随着浮点处理器的出现，为避免特征值分解而进行数学上的简化，意义已经不大。因此，可以通过显式地计算特征值来检测Harris角点。原则上这种修改不会明显地影响检测结果，但是可以避免使用随意的k参数。有两个函数可以用来显式地计算Harris协方差矩阵的特征值（以及特征向量），即cv::cornerEigenValsAndVecs和cv::cornerMinEigenVal。

第二项改进是针对特征点聚集的问题。事实上，尽管引入了局部最大值这个条件，兴趣点仍会在图像中不均匀分布，聚集在高度纹理化的位置。解决该问题的一种方案，就是限制两个兴趣点之间的最短距离。可以通过下面的算法实现。从Harris值最强的点开始（即具有最大的最低特征值），只允许一定距离之外的点成为兴趣点。在OpenCV中用函数cv::goodFeaturesToTrack实现这个算法。之所以采用这个函数名称，是因为它检测的特征非常适合作为视觉跟踪程序的起始集合。调用方法如下：

```
// 计算适合跟踪的特征
std::vector<cv::Point2f> corners;
cv::goodFeaturesToTrack(image, // 输入图像
   corners, // 角点图像
   500,     // 返回角点的最大数量
   0.01,    // 质量等级
   10);     // 角点之间允许的最短距离
```

除了作为质量等级的阈值和兴趣点之间允许的最短距离，该函数还使用了返回角点的最大数量（角点是按照强度排序的，因此这种做法可行）。上述调用后得到以下结果：



由于需要对兴趣点按照Harris评分排序，因此增加了该方法检测的复杂度，但是它也明显地改进了兴趣点在整幅图像中的分布情况。注意这个函数还有一个可选的标志，该标志要求在检测Harris角点时，采用经典的角点评分定义（使用协方差矩阵的行列式值和迹）。

2. 特征检测方法的通用接口

OpenCV 2为各种兴趣点检测方法引入了一个通用的接口。可以很方便地在同一程序中用这个接口测试各种兴趣点检测方法。

该接口定义了`cv::Keypoint`类，封装每个被检测特征点的属性。对于Harris角点而言，有关的属性只有关键点位置和它的强度。8.4节将讨论与关键点有关的其他属性。

抽象类`cv::FeatureDetector`要求它的继承类必须有`detect`方法，签名如下：

```
void detect( const Mat& image, vector<KeyPoint>& keypoints,
             const Mat& mask=Mat() ) const;

void detect( const vector<Mat>& images,
             vector<vector<KeyPoint> >& keypoints,
             const vector<Mat>& masks=
                               vector<Mat>() ) const;
```

第二个方法可以在图像向量中检测兴趣点。这个类还包含其他方法，可以从文件读写被检测的兴趣点。

`cv::goodFeaturesToTrack`函数被封装在`cv::GoodFeaturesToTrackDetector`类中，它是从`cv::FeatureDetector`类继承的。它的用法与Harris角点类差不多，如下所示：

```
 // KeyPoint类型的向量
 std::vector<cv::KeyPoint> keypoints;
 // 适合跟踪的特征检测器的构造函数
cv::Ptr<cv::FeatureDetector> gftt=
   new cv::GoodFeaturesToTrackDetector(
   500,  // 返回角点的最大数量
   0.01, // 质量等级
   10);  // 兴趣点之间允许的最短距离
// 用FeatureDetector方法检测兴趣点
gftt->detect(image,keypoints);
```

得到的结果与前面相同，这是因为封装成类后，最终调用的函数是一样的。注意这里我们用了OpenCV 2的智能指针类（`cv::Ptr`），它会在引用数降为0时自动释放指针对象，第1章对此做了解释。

8.2.4　参阅

- C. Harris和M.J. Stephens发表在*Alvey Vision Conference* 1988年第147页至第152页的

“A combined corner and edge detector”是描述Harris算子的经典论文。

- J. Shi和C. Tomasi发表在*Int. Conference on Computer Vision and Pattern Recognition* 1994年第593页至第600页的论文“Good features to track”介绍了这些特征。
- K. Mikolajczyk和C. Schmid发表在*International Journal of Computer Vision* 2004年第60卷第1期第63页至第86页的论文“Scale and Affine invariant interest point detectors”提出了多尺度和仿射不变的Harris算子。

8.3　快速检测特征

Harris算子对角点（或者更通用的兴趣点）作出了规范的数学定义，该定义基于在两个互相垂直的方向上强度值的变化率。虽然这是一种很完美的定义，但它需要计算图像的导数，而计算导数是非常耗时的。尤其要注意的是，检测兴趣点通常只是更复杂算法中的第一个步骤。

本节我们介绍另一种特征点算子，即FAST（加速分割测试获得特征，Features from Accelerated Segment Test）。这种算子专门用来快速检测兴趣点；只需要对比几个像素，就可以判断是否为关键点。

8.3.1　如何实现

由于OpenCV2有检测特征点的通用接口，因此调用任何特征点检测器都非常容易。本节介绍的是FAST检测器。顾名思义，它的设计目的就是进行快速计算：

```
// 关键点的向量
std::vector<cv::KeyPoint> keypoints;
// FAST特征检测器的构造函数
cv::Ptr<cv::FeatureDetector> fast=
new cv::FastFeatureDetector(
  40); // 检测用的阈值
// 检测特征点
fast->detect(image,keypoints);
```

OpenCV也提供了在图像上画关键点的通用函数：

```
cv::drawKeypoints(image,    // 原始图像
   keypoints,               // 关键点的向量
   image,                   // 输出图像
   cv::Scalar(255,255,255), // 关键点的颜色
   cv::DrawMatchesFlags::DRAW_OVER_OUTIMG); // 画图标志
```

选择这个画图标志后，会在输入图像上画出关键点，输出结果如下：



有一种比较有趣的做法，就是用一个负数作为关键点颜色。这样一来，画每个圆时会随机选用不同的颜色。

8.3.2 实现原理

跟Harris检测器的情况一样，FAST算法源于对构成角点的定义。FAST对角点的定义基于候选特征点周围的图像强度值。以某个点为中心作一个圆，根据圆上的像素值判断该点是否为关键点。如果存在这样一段圆弧，它的连续长度超过周长的3/4，并且它上面所有像素的强度值都与圆心的强度值明显不同（全部更黑或更亮），那么就认定这是一个关键点。

这种测试方法非常简单，计算速度很快。而且在它的原始公式中，算法还用了一个技巧来进一步提高处理速度。如果首先测试圆周上相隔90度的四个点（例如取上、下、左、右四个位置），很容易证明：为了满足前面的条件，必须至少有其中的三个点，都比圆心更亮或都比圆心更黑。

如果不满足该条件，那就可以立即排除这个点，不需要检查圆周上的其他点。这种方法非常高效，因为在实际应用中，图像中大部分像素都可以用这种“四点比较”法排除。

从概念上讲，用于检查像素的圆的半径应该作为方法的一个参数。但是根据经验，半径为3时可以得到好的结果和较高的计算效率。因此需要在圆周上检查16个像素，如下图所示：

		16	1	2		
	15				3	
14						4
13			0			5
12						6
	11				7	
		10	9	8		

用来预测试的像素是1、5、9和13，比圆心更黑或更亮的最少连续像素数是12。但是根据经验，把连续线段长度减少到9，可以使对图像角点的检测有更好的可重复性。这一变种称为**FAST-9**角点检测器，并且它就是OpenCV采用的方法。注意，有一个`cv::FAXTX`函数，用于另一种FAST检测器。

一个点与圆心的强度值的差距必须达到一个指定的值，才被认为明显更黑或更亮；这个值就是调用函数时指定的阈值参数。这个阈值越大，检测到的角点数量就越少。

至于Harris特征，通常最好在发现的角点上执行非最大值抑制。因此，需要定义一个角点强度的衡量方法。有多种衡量方法可供选择，下面介绍的是实际选用的方法。计算中心点像素与认定的连续圆弧上的像素的差值，然后将这些差值的绝对值累加，就得到角点强度。要使用该算法，可以直接调用函数：

```
cv::FAST(image,    // 输入图像
        keypoints, // 输出关键点的向量
        40,        // 阈值
        false);    // 是否进行非最大值抑制?
```

但是因为它比较复杂，推荐使用`cv::FeatureDetector`接口。

用这个算法检测兴趣点的速度非常快，因此十分适合需要优先考虑速度的应用。这些应用包括实时视觉跟踪、目标识别等，它们需要在实时视频流中跟踪或匹配多个点。

8.3.3　扩展阅读

为了更好地检测特征点，OpenCV提供了一些附加工具。事实上，有很多类适配器，可更好地控制提取关键点的方式。

1. 适配的特征检测

如果要更好地控制被检测特征点的数量，可使用`cv::FeatureDetector`的一个专门的子

类，即cv::DynamicAdaptedFeatureDetector。它可以指定被检测兴趣点的数量范围。这时使用FAST特征检测器的方法如下：

```
cv::DynamicAdaptedFeatureDetector fastD(
  new cv::FastAdjuster(40), // 特征检测器
  150,   // 特征的最小数量
  200,   // 特征的最大数量
  50);   // 最大迭代次数
fastD.detect(image,keypoints); // 检测特征点
```

这段代码会采用迭代方式检测兴趣点。每次迭代后都会检查兴趣点数量，并且调整阈值以增加或减少数量。这个过程会反复进行，直到被检测兴趣点的数量符合指定的范围。为了避免多次检测浪费太多时间，我们指定了最大迭代次数。如果用常规方式实现这个方法，则必须在cv::FeatureDetector类中实现cv::AdjusterAdapter接口。这个类包含了一个tooFew方法和一个tooMany方法，两个方法都会修改内部阈值，以便生成更多或更少的关键点。此外还有一个good断言方法，如果检测器的阈值还能调节，它就返回true。要获得适当数量的特征点，采用cv::DynamicAdaptedFeatureDetector类是不错的办法。但是必须明白，它带来好处的同时，必须付出性能上的代价。而且在指定的迭代次数内，并不保证能实际得到所需数量的特征点。

也许你已经注意到，传入的参数中有一个动态分配对象的地址，指定了即将在匹配类中使用的特征检测器。你可能想知道，是否必须在某处释放分配的内存，以避免内存泄漏。答案是不需要，这是因为这个指针被传送到一个cv::Ptr<FeatureDetector>类型的参数，它会自动释放指向的对象。

2. 网格适配特征检测

第二个要使用的适配类是cv::GridAdaptedFeatureDetector。顾名思义，可以使用它在图像上定义一个网格。网格中的每个单元格可以设置一个最大元素数量。这里的理念是把被检测的关键点以更好的方式扩展到整幅图像上。实际上在检测图像中的关键点时，经常出现这种情况，即很多兴趣点聚集在特定的纹理区域。例如在前面的教堂图中两个塔楼的位置就检测到了非常密集的FAST点。这个适配类的用法如下：

```
cv::GridAdaptedFeatureDetector fastG(
  new cv::FastFeatureDetector(10), // 特征检测器
  1200,  // 关键点总数的最大值
  5,     // 网格的行数
  2);    // 网格的列数
fastG.detect(image,keypoints);
```

这个适配类的实现方法是用cv::FeatureDetector对象在每个独立的单元格中检测特征点。同时指定了总数的最大值。在每个单元格中，为了不超出最大限值，只保留了最强的特征点。

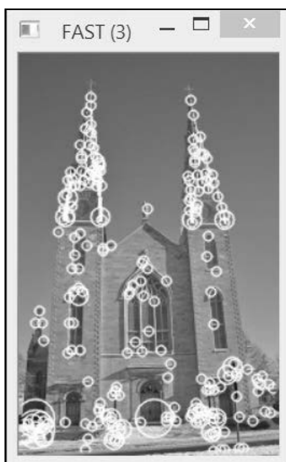
3. 金字塔适配特征检测

适配类cv::PyramidAdaptedFeatureDetector的做法是在一个图像金字塔上应用特征

检测器。结果被合并到输出的关键点向量中。调用方式如下：

```
cv::PyramidAdaptedFeatureDetector fastP(
  new cv::FastFeatureDetector(60), // 特征检测器
  3);    // 金字塔的层数
fastP.detect(image,keypoints);
```

每个点的坐标在原始图像的坐标中指定。并且设置了`cv::Keypoint`类的专用属性`size`，设置这个属性后，如果某个图层的分辨率是原始图像的一半，那么在该图层上检测到的特征点的尺寸将会是原始图像上特征点的两倍。使用`cv::drawKeypoints`函数中一个特殊标志后，画关键点时用的半径，等于该关键点的`size`属性。



8.3.4　参阅

- E. Rosten和T. Drummond发表在*European Conference on Computer Vision* 2006年第430页至第443页的“Machine learning for high-speed corner detection”详细描述了FAST特征算法和它的变种。

8.4　尺度不变特征的检测

8.1节讲过，特征检测的视觉不变性是一个非常重要的概念。前面介绍的特征检测器已经可以较好地解决方向不变性问题，即图像旋转后仍能检测到相同的特征点。但是要解决尺度不变性问题，难度就大多了。为解决这一问题，计算机视觉界引入了尺度不变特征的概念。它的理念是，不仅在任何尺度下拍摄的物体都能检测到一致的关键点，而且每个被检测的特征点都对应一个尺度因子。理想情况下，对于两幅图像中不同尺度的的同一个物体点，计算得到的两个尺度因子之间的比率应该等于图像尺度的比率。近几年，人们提出了多种尺度不变特征，本节介绍其中的一

种：SURF特征。SURF全称为“加速稳健特征”（Speeded Up Robust Feature），我们将会看到，它们不仅是尺度不变特征，而且是具有较高计算效率的特征。

8.4.1　如何实现

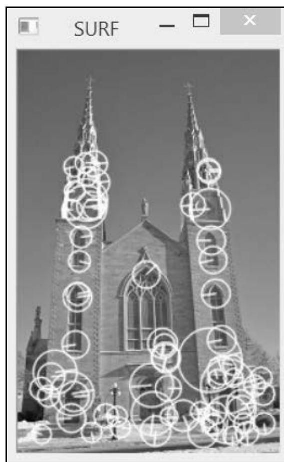
OpenCV的函数`cv::SURF`实现了SURF特征的检测。也可以通过`cv::FeatureDetector`使用这个函数，代码如下：

```
// SURF特征检测器的构造函数
cv::Ptr<cv::FeatureDetector> detector =
  new cv::SURF(2000.); // 阈值
// 检测SURF特征
detector->detect(image,keypoints);
```

为了画出这些特征，我们仍使用OpenCV的`cv::drawKeypoints`函数，并且采用`DRAW_RICH_KEYPOINTS`标志以便显示相关的尺度因子:

```
// 画出关键点，包括尺度和方向信息
cv::drawKeypoints(image,      // 原始图像
   keypoints,                 // 关键点的向量
   featureImage,              // 结果图像
   cv::Scalar(255,255,255),   // 点的颜色
   cv::DrawMatchesFlags::DRAW_RICH_KEYPOINTS); //标志
```

包含被检测特征的结果图像如下：



上一节解释过，使用`DRAW_RICH_KEYPOINTS`标志可得到关键点的圆，并且圆的尺寸与每个特征计算得到的尺度成正比。为了使特征具有旋转不变性，SURF还让每个特征关联了一个方向。方向由每个圆内的一条辐射线表示。

如果对同一个物体，用不同的尺度拍另一张照片，特征检测的结果如下：



仔细观察两幅图像中的关键点，可以发现圆的大小变化与尺度的变化总是成正比的。举个例子，通过观察教堂右上角窗口的底部，可以看出两幅图像都在这个位置检测到了SURF特征，并且对应的圆（大小不同）包含了同样的视觉元素。当然并不是全部特征都如此，但是正如下一章将揭示的，此时的重复率已经高到可以使两幅图像很好地匹配。

8.4.2　实现原理

在第6章我们学过，可以用高斯滤波器估算图像的导数。高斯滤波器用σ参数定义内核的口径（尺寸）。我们知道，这个σ参数对应了用于构建滤波器的高斯函数的变化幅度，它还隐式地定义了计算导数的范围。事实上，滤波器的σ值越大，图像的细节越平滑。因此它可以在更粗糙的范围内操作。

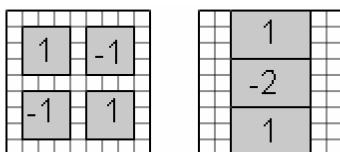
如果在不同的尺度内，用高斯滤波器计算指定像素的拉普拉斯算子，会得到不同的数值。观察滤波器对不同尺度因子的响应规律，可以得到一条具有最大σ值的曲线。对于以不同尺度拍摄的两幅图像的同一个物体，对应的两个σ值的比率等于拍摄两幅图像的尺度的比率。这一重要观察是尺度不变特征提取过程的核心。也就是说，为了检测尺度不变特征，需要在图像空间（图像中）和尺度空间（通过在不同尺度下应用导数滤波器得到）分别计算局部最大值。

SURF用以下方法实现了这个理论。首先，为了检测特征而对每个像素计算Hessian矩阵。该矩阵衡量了一个函数的局部曲率，定义如下：

$$\boldsymbol{H}(x,y)=\begin{bmatrix}\dfrac{\delta^2 I}{\delta x^2} & \dfrac{\delta^2 I}{\delta x\delta y}\\[2ex] \dfrac{\delta^2 I}{\delta x\delta y} & \dfrac{\delta^2 I}{\delta y^2}\end{bmatrix}$$

根据矩阵的行列式值，可以得到曲率的强度。该方法把角点定义为局部高曲率（即在多个方向上的变化幅度都很高）的像素点。这个矩阵由二阶导数构成，因此可以用高斯内核的拉普拉斯算子，在不同的尺度（例如σ）下计算得到。这样Hessian矩阵就成了三个变量的函数，即$H(x, y, \sigma)$。如果在普通空间和尺度空间（即需要执行3 × 3× 3次非最大值抑制），Hessian矩阵的行列式值达到了局部最大值，那么就认为这是一个尺度不变特征。注意，为了确认点的有效性，必须在`cv::SURF`构造函数的第一个参数中指定最小行列式值。

但是在不同尺度下计算全部导数值，计算量非常大。SURF算法的目标是使这个过程尽可能地高效。具体做法是使用近似的高斯内核，只附带几个整数。它们的结构如下：



左边的内核用于估算混合二阶导数，右边的内核用于估算垂直方向的二阶导数。右边的内核旋转后，就可估算水平方向的二阶导数。最小的内核尺寸为9 × 9像素，对应$\sigma \approx 1.2$。要在尺度空间中使用，需要连续地应用一系列内核，并且内核的尺寸逐个增大。可以在SURF类的附加参数中指定滤波器的准确数量。默认使用12个不同尺寸的内核（最大尺寸为99 × 99）。该算法采用积分图像，这是为了确保只用三个加法运算就可以计算每个滤波器分支的累加值，与滤波器尺寸无关。

一旦找到局部最大值，就可以使用尺度空间和图像空间的插值法，获得被检测兴趣点的精确位置。得到的一批位于子像素级精度的特征点，并且每个特征点关联了一个尺度值。

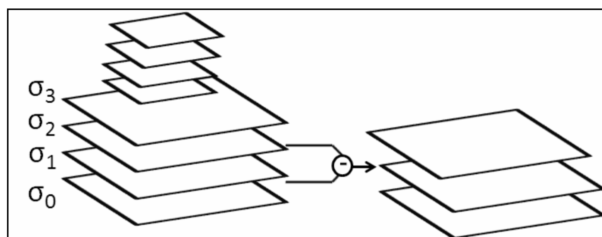
8.4.3　扩展阅读

SURF算法是SIFT算法的加速版，而SIFT（尺度不变特征转换，Scale-Invariant Feature Transform）是另一种著名的尺度不变特征检测法。

SIFT特征检测算法

SIFT检测特征时，也采用图像空间和尺度空间的局部最大值，但它使用拉普拉斯滤波器响应，而不是Hessian行列式值。这个拉普拉斯算子是利用高斯滤波器的差值，在不同尺度（即逐步加大σ值）下计算得到的，详见第6章。为了提高性能，每次σ值增加一倍，图像的尺寸就缩小一半。每个金字塔级别代表一个八度（octave），每个尺度是一图层（layer）。一个八度通常包含三个图层。

下图表示两个八度的金字塔，其中第一个八度的四个高斯滤波图像产生三个DoG图层：



OpenCV中有一个用于检测这些特征的类，它的调用方法与SURF类似：

```
// 构造SIFT特征检测器对象
detector = new cv::SIFT();
// 检测SIFT特征
detector->detect(image,keypoints);
```

这里构造函数的参数都用了缺省值，但是也可以指定所需的SIFT点数量（保留强度最大的点）、每个八度包含的图层数以及σ的初始值。得到的结果与SURF类似：



然而，由于SIFT基于浮点内核计算特征点，因此通常认为，SIFT算法检测的特征在空间和尺度上定位更加精确。基于同样的原因，它的计算效率也更低，尽管相对效率取决于具体的实现方法。

最后提醒一下，也许你已经注意到，SURF和SIFT类被放在非免费的OpenCV软件包中。因为这几种算法受专利保护，在商业程序中使用会受到许可协议的限制。

8.4.4　参阅

- 6.5节详细介绍了拉普拉斯–高斯算子和不同高斯差的应用。
- 9.3节解释了如何在稳健图像匹配中使用尺度不变特征。

- H. Bay，A. Ess、T. Tuytelaars和L. Van Gool发表在*Computer Vision and Image Understanding* 2008年第110卷第3期第346页至第359页的“SURF: Speeded Up Robust Features”描述了SURF算法。
- D. Lowe发表在*International Journal of Computer Vision* 2004年第60卷第2期第91页至第110页的“Distinctive Image Features from Scale Invariant Features”描述了SIFT算法。

8.5 多尺度 FAST 特征的检测

FAST是一种快速检测图像关键点的方法。使用SURF和SIFT算法时，侧重点在于设计尺度不变特征。后来引入了新的兴趣点检测方法，既能快速检测，又不随尺度改变而变化。本节介绍BRISK（Binary Robust Invariant Scalable Keypoints，二元稳健恒定可扩展关键点）检测法。它基于上节介绍的FAST特征检测法。本节最后还将讨论另一种检测方法，称为ORB（Oriented FAST and Rotated BRIEF，定向FAST和旋转BRIEF）。在需要快速可靠地匹配图像时，使用这两种特征点检测法是非常优秀的解决方案。在结合相关的二值描述子使用时，它们的性能特别高，在第9章将详细讨论。

8.5.1 如何实现

依据上节的内容，BRISK算法使用抽象类`cv::FeatureDetector`检测关键点。首先创建该检测器的实例，然后调用`detect`方法：

```
// 构造BRISK特征检测器对象
detector = new cv::BRISK();
// 检测BRISK特征
detector->detect(image,keypoints);
```

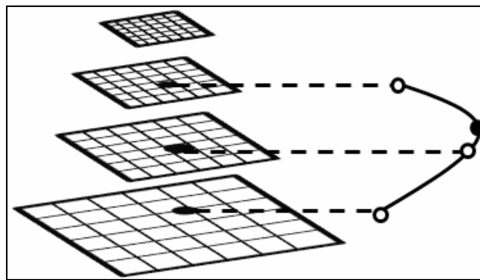
下图显示了在多个尺度下检测到的关键点：



8.5.2　实现原理

BRISK不仅是一个特征点检测器，它还包含了描述每个被检测关键点的邻域的过程。第二部分的内容就是下一章的主题。我们将讨论如何用BRISK算法在多个尺度下快速地检测关键点。

为了在不同尺度下检测兴趣点，该算法首先通过两个下采样过程构建一个图像金字塔。第一个过程从原始图像尺寸开始，然后每一图层（八度）减少一半。第二个过程首先将原始图像的尺寸除以1.5，得到第一个图像，然后在这个图像的基础上，每一层减少一半，两个过程产生的图层交替在一起。



然后在该金字塔的所有图像上应用FAST特征检测器。提取关键点的条件与SIFT算法类似。首先，一个像素与八个相邻像素之一比较强度值，只有是局部最大值的像素才可能称为关键点。这个条件满足后，这个点与上下两层的相邻像素比较评分；如果它的评分在尺度上也更高，那么就认为它是一个兴趣点。BRISK算法的关键在于：金字塔的各个图层具有不同的分辨率。为了精确定位每个关键点，算法需要在尺度和空间两个方面进行插值。插值基于FAST关键点评分。在空间方面，在3×3的邻域上进行插值。在尺度方面，计算依据是要适合一个一维抛物线，该抛物线在尺度坐标轴上，并穿过当前点和上下两层的两个相邻的局部关键点；这个关键点在尺度上的位置，见前面的图片。这样做的结果是，即使在不连续的图像尺度上执行FAST关键点检测，最后检测到每个关键点对应的尺度都是连续的值。

`cv::BRISK`类有两个可选参数，用于控制关键点的检测。第一个参数是判断FAST关键点的阈值，第二个参数是图像金字塔中生成的八度的数量：

```
// 构造另一个BRISK特征检测器对象
detector = new cv::BRISK(
  20,  // 判断是否为FAST点的阈值
  5);  // 八度的数量
```

8.5.3　扩展阅读

在OpenCV中，BRISK并不是唯一的多尺度快速检测器。ORB特征检测器也能进行快速关键点检测。

ORB特征检测算法

ORB代表定向FAST和旋转BRIEF（Oriented FAST and Rotated BRIEF）。这个缩写的第一层意思表示关键点检测，第二层意思表示ORB算法提供的描述子。这里我们关注检测方法，下一章将介绍描述子。

跟BRISK一样，ORB首先创建一个图像金字塔。它由一批图层组成，每个图层都是用固定的缩放因子对前一个图层下采样得到（典型情况是用8个尺度，缩放因子为1.2。这些参数可在cv::ORB函数中设置）。在具有关键点评分的位置，接受N个强度最大的关键点。其中关键点评分用的是8.2节定义的Harris角点强度衡量方法（这个方法的作者发现Harris评分是更可靠的衡量方法）。

ORB检测器的原理基于一个现象，即每个被检测的兴趣点总是关联了一个方向。下一章我们将看到，这个信息可用于校准不同图像中检测的关键点描述子。在7.6节，我们介绍了图像轮廓矩的概念，并且特别展示了如何用前三个轮廓矩计算区域的重心。ORB算法建议使用关键点周围的圆形邻域的重心的方向。因为根据定义，FAST关键点肯定有一个偏离中心点的重心，中心点与重心的连线的角度总是非常明确的。

ORB特征的检测方法如下：

```
// 构造ORB特征检测器对象
detector = new cv::ORB(200, // 关键点的总数
                          1.2, // 图层之间的缩放因子
                    8); // 金字塔的图层数量
// 检测ORB特征
detector->detect(image,keypoints);
```

调用的结果如下：



可以发现，因为金字塔中每个图层的关键点是独立检测的，检测器会在不同尺度中重复检测同一个特征点。

8.5.4 参阅

- 9.4节将解释如何用简单的二元描述子快速稳健地匹配这些特征。
- S. Leutenegger、M. Chli和R. Y. Siegwart发表在*IEEE International Conference on Computer Vision* 2011年第2448页至第2555页的“BRISK: Binary Robust Invariant Scalable Keypoint”描述了BRISK特征算法。
- E. Rublee、V. Rabaud、K. Konolige和G. Bradski发表在*IEEE International Conference on Computer Vision* 2011年第2564页至第2571页的“ORB: an efficient alternative to SIFT or SURF”描述了ORB特征算法。

第 9 章

描述和匹配兴趣点



本章包括以下内容：

- 局部模板匹配；
- 描述局部强度值模式；
- 用二值特征描述关键点。

9.1 简介

上一章我们学习了如何检测图像中的特殊点集，以便进行后续的局部图像分析。这些关键点都具有足够的独特性，如果一个物体在一个图像中被检测到关键点，那么同一个物体在其他图像中也会检测到同一个关键点。我们还描述了几个更复杂的兴趣点检测器，它们可以在关键点上设置有代表性的缩放因子和（或）方向。我们将在本节中看到，这个额外的信息可用于规范不同视角的场景展示。

为了进行基于兴趣点的图像分析，我们需要构建能够唯一地描述关键点的展现方式。本章将探讨从兴趣点提取描述子的各种方法。这些描述子通常是二值类型、整数型或浮点数型组成的一维或二维向量，描述了一个关键点和它的邻域。好的描述子要具有足够的独特性，能唯一地表示图像中每个关键点。它还要有足够的鲁棒性，在照度变化或视角变动时仍能较好地体现同一批点集。理想的描述子还要简洁，便于处理。

图像匹配是关键点的常用功能之一。它的作用包括关联同一场景的两个图像、检测图像中事物的发生地点，等等。本章我们来学习几种基本的匹配策略，下一章将更深入地讨论。

9.2 局部模板匹配

通过特征点匹配，可以将一幅图像的点集和另一幅图像（或一批图像）的点集关联起来。如果两个点集对应着现实中的同一个场景元素（或物体点），它们就应该匹配。

要判断两个关键点的相似度，仅凭单个像素显然是不够的。因此需要在匹配过程中考虑每个关键点周围的图像块。如果两个图像块对应着同一个场景元素，那么它们的像素值应该会比较相似。本节介绍的方案是对图像块中的像素进行逐个比较。这可能是最简单的特征点匹配方法了，但是我们将会看到，这种方法并不是最可靠的。不过在某些情况下，它也能得到不错的结果。

9.2.1　如何实现

最常见的图像块是边长为奇数的正方形，关键点位置就是正方形的中心。可通过比较块内像素的强度值，来衡量两个正方形图像块的相似度。常见的方案是采用简单的*差的平方和*（Sum of Squared Differences，SSD）算法。下面是特征匹配策略的具体步骤。首先要检测每个图像的关键点。这里我们使用FAST检测器：

```
// 定义关键点向量
std::vector<cv::KeyPoint> keypoints1;
std::vector<cv::KeyPoint> keypoints2;
// 定义特征点检测器
cv::FastFeatureDetector fastDet(80);
// 检测特征点
fastDet.detect(image1,keypoints1);
fastDet.detect(image2,keypoints2);
```

然后定义一个11×11的矩形，表示每个关键点周围的图像块：

```
// 定义正方形的邻域
const int nsize(11); // 邻域的尺寸
cv::Rect neighborhood(0, 0, nsize, nsize); // 11×11
cv::Mat patch1;
cv::Mat patch2;
```

一幅图像的关键点与另一幅图像的全部关键点进行比较。对于第一幅图像的每个关键点，找出在第二幅图像中与它最相似的图像块。这个过程用两个嵌套循环实现，代码如下：

```
// 针对第一幅图像中的每个关键点，在第二幅图像中找出最匹配的
cv::Mat result;
std::vector<cv::DMatch> matches;

// 针对图像一的全部关键点
for (int i=0; i<keypoints1.size(); i++) {

   // 定义图像块
   neighborhood.x = keypoints1[i].pt.x-nsize/2;
   neighborhood.y = keypoints1[i].pt.y-nsize/2;

   // 如果邻域超出图像范围，就继续处理下一个点
   if (neighborhood.x<0 || neighborhood.y<0 ||
       neighborhood.x+nsize >= image1.cols ||
           neighborhood.y+nsize >= image1.rows)
       continue;
```

```
    //图像一的块
    patch1 = image1(neighborhood);

    // 复位最匹配的值
    cv::DMatch bestMatch;

    // 针对图像二的全部关键点
    for (int j=0; j<keypoints2.size(); j++) {

        // 定义图像块
        neighborhood.x = keypoints2[j].pt.x-nsize/2;
        neighborhood.y = keypoints2[j].pt.y-nsize/2;

        // 如果邻域超出图像范围，就继续处理下一个点
        if (neighborhood.x<0 || neighborhood.y<0 ||
          neighborhood.x + nsize >= image2.cols ||
               neighborhood.y + nsize >= image2.rows)
            continue;

      // 图像二的块
      patch2 = image2(neighborhood);

      // 匹配两个图像块
         cv::matchTemplate(patch1,patch2,result,
                           CV_TM_SQDIFF_NORMED);

      // 检查是否最佳的匹配
      if (result.at<float>(0,0) < bestMatch.distance) {

        bestMatch.distance= result.at<float>(0,0);
        bestMatch.queryIdx= i;
        bestMatch.trainIdx= j;
      }
    }

    // 添加最佳匹配
    matches.push_back(bestMatch);
}
```

注意，这里用了`cv::matchTemplate`函数来计算图像块的相似度（下一节将详细介绍这个函数）。找到一个可能的匹配项后，用一个`cv::DMatch`对象来表示。该对象存储了两个被匹配关键点的序号和相似度。

两个图像块越相似，它们对应着同一个场景点的可能性就越大。因此需要根据相似度对匹配结果进行排序：

```
// 提取25个最佳匹配项
std::nth_element(matches.begin(),
                 matches.begin()+25,matches.end());
matches.erase(matches.begin()+25,matches.end());
```

你可以用一个相似度阈值筛选这些匹配项，并得出筛选结果。这里我们保留相似度最大的N个匹配项（为了方便显示结果，选用$N=25$）。

有趣的是，OpenCV本身就带有一个能显示匹配结果的函数，它把两个图像拼接起来，然后用线条连接每个对应的点。函数的用法如下：

```
// 画出匹配结果
cv::Mat matchImage;
cv::drawMatches(image1,keypoints1,          // 图像一
                image2,keypoints2,          // 图像二
                matches,                    // 匹配项的向量
                cv::Scalar(255,255,255),    // 线条颜色
                cv::Scalar(255,255,255));   // 点的颜色
```

得到的结果如下：



9.2.2　实现原理

这样的结果显然并不理想，但是通过观察这些点集的匹配效果，也能发现一些成功的匹配项。还可以发现建筑物的重复结构造成了一些混淆。另外，我们试图为左侧图像的所有点集，在右侧找到与它匹配的点集，因此出现了右侧点集与多个左侧点集匹配的情况。有一些方法可以修正这种不对称的匹配项，例如对右侧点集只保留相似度最大的匹配项。

这里我们用一个简单的标准来比较图像块，即指定CV_TM_SQDIFF标志，逐个像素地计算差值的平方和。在比较图像I_1的像素(x, y)和I_2的像素（x', y'）时，用下面的公式衡量相似度：

$$\sum_{i,j}(I_1(x+i, y+j)-I_2(x'+i, y'+j))^2$$

这些(i, j)点的累加值，就是以每个点为中心的整个正方形模板的偏移值。如果两个图像块比较相似，它们的相邻像素之间的差距就比较小，因此累加值最小的块就是最匹配的图像块。该功

能通过匹配函数的主循环实现，即针对一个图像的每个关键点，在另一个图像中找出差值平方和最小的关键点。也可以设置一个阈值，排除掉差值平方和超过该阈值的匹配项。在本例中只是将结果按照相似度从大到小进行排序。

这个例子用11×11的方块进行匹配。采用更大的邻域会使图像块更具独特性，但是也会导致对局部的场景变化更加敏感。

只要两幅图像所表现场景的视角和拍摄条件都比较相似，简单地用差值平方和来比较两个图像窗口也能得到较好的效果。实际上只要光照有变化，就会增强或降低图像块中所有像素的强度值，导致差值平方发生很大的变化。为了减少光照对匹配结果的影响，还可采用衡量图像窗口相似度的其他公式。OpenCV提供了很多这样的公式，其中归一化的差值平方和（用`CV_TM_SQDIFF_NORMED`标志）非常实用：

$$\frac{\sum_{i,j}(\boldsymbol{I}_1(x+i,y+j)-\boldsymbol{I}_2(x'+i,y'+j))^2}{\sqrt{\sum_{i,j}\boldsymbol{I}_1(x+i,y+j)^2}\sqrt{\sum_{i,j}\boldsymbol{I}_2(x'+i,y'+j)^2}}$$

其他相似度衡量方法基于信号处理理论中的相关性，定义如下（用`CV_TM_CCORR`标志）：

$$\sum_{i,j}(\boldsymbol{I}_1(x+i,y+j)\boldsymbol{I}_2(x'+i,y'+j)$$

如果两个图像块非常相似，这个值将达到最大。

识别出的匹配项存储在`cv::DMatch`类型的向量中。本质上`cv::DMatch`数据结构的内部包含两个索引，第一个索引指向第一个关键点向量中的元素，第二个索引指向第二个关键点向量中匹配上的特征点。它还包含一个数值，表示两个已匹配的描述子之间的差距。运算符<可用于比较两个`cv::DMatch`实例，它的定义中用到了这个差距值。

为了使结果更具可读性，在绘制匹配项时要限制线条的数量。因此，我们只显示了差距最小的25个匹配项。要想实现这个功能，需要调用函数`std::nth_element`。这个函数将排序后的第N个元素放在第N个位置，比这个元素小的元素放在它的前面，然后把向量中其余的元素清除。

9.2.3　扩展阅读

在这个特征点检测方法中，函数`cv::matchTemplate`是关键。这里我们采用非常特殊的方式调用它，即用它来比较两个图像块。但是这个函数本身是很通用的。

模板匹配

在图像分析中，一个常见的任务是检测图像中是否存在特定的图案或物体。实现方法是把包

含该物体的小图像作为模板，然后在指定图像上搜索与模板相似的部分。搜索的范围通常仅限于可能发现该物体的区域。在这个区域上滑动模板，并在每个像素的位置计算相似度。执行这个操作的函数是cv::matchTemplate。函数的输入对象是一个小图像模板和一个被搜索的图像。结果是一个浮点数型的cv::mat函数，表示每个像素位置上的相似度。假设模板尺寸为$M \times N$，图像尺寸为$W \times H$，那么结果矩阵的尺寸是$(W-N+1)\times(H-N+1)$。通常我们只关注相似度最高的位置。典型的模板匹配代码如下（假设目标变量就是这个模板）：

```
// 定义搜索区域
cv::Mat roi(image2,
  // 这里用图像的上半部分
  cv::Rect(0,0,image2.cols,image2.rows/2));

// 进行模板匹配
cv::matchTemplate(
  roi,     // 搜索区域
  target, // 模板
  result, // 结果
  CV_TM_SQDIFF); // 相似度

// 找到最相似的位置
double minVal, maxVal;
cv::Point minPt, maxPt;
cv::minMaxLoc(result, &minVal, &maxVal, &minPt, &maxPt);

// 在相似度最高的位置绘制矩形
// 本例中为minPt
cv::rectangle(roi,
   cv::Rect(minPt.x, minPt.y, target.cols , target.rows),
   255);
```

要知道这个操作是非常耗时的，因此应该限制搜索的区域，并且模板的像素要少。

9.2.4　参阅

- 下一节介绍本节中实现匹配策略的cv::BFMatcher类。

9.3　描述局部强度值模式

第8章讨论的SURF和SIFT关键点检测算法，定义了每个被检测特征的位置、方向和尺度。在定义分析特征点的窗口大小时，要用到尺度因子的信息。因此不管该特征所属物体的拍摄比例是多大，定义的邻域都将包含同样的视觉信息。本节将介绍如何用特征描述子来描述兴趣点的邻域。在图像分析中，可以用邻域包含的视觉信息来标识每个特征点，以便区分各个特征点。特征描述子通常是一个N维的向量，在光照变化和拍摄角度微小扭曲时，它描述特征点的方式不会发生变化。通常可以用简单的差值矩阵来比较描述子，例如用欧几里德距离。在特征匹配应用中，特征

描述子是一种强大的工具。

9.3.1 如何实现

OpenCV 2提供了一个通用的接口，用于批量计算关键点的描述子。该接口名为cv::DescriptorExtractor，它的使用方法与上一章使用cv::FeatureDetector接口的方法类似。大多数基于特征的方法都包含一个检测器和一个描述子组件；cv::SURF和cv::SIFT等类同时实现了这两个接口。这意味着在检测和描述关键点时只需要创建一个对象。匹配两个图像的方法如下：

```
// 定义特征检测器
// 构造SURF特征检测器对象
cv::Ptr<cv::FeatureDetector> detector = new cv::SURF(1500.);

// 关键点检测
// 检测SURF特征
detector->detect(image1,keypoints1);
detector->detect(image2,keypoints2);

// SURF同时包含了检测器和描述子提取器
cv::Ptr<cv::DescriptorExtractor> descriptor = detector;

// 提取描述子
 cv::Mat descriptors1;
 cv::Mat descriptors2;
 descriptor->compute(image1,keypoints1,descriptors1);
 descriptor->compute(image2,keypoints2,descriptors2);
```

对于SIFT，只需改成创建cv::SIFT()对象。函数返回一个矩阵（即cv::Mat实例），矩阵的行数等于关键点向量的元素个数。每行是一个N维的描述子向量。对于SURF描述子，它的默认尺寸是64；而对于SIFT，默认尺寸是128。这个向量用于区分特征点周围的强度值图案。两个特征点越相似，它们的描述子向量就会越接近。

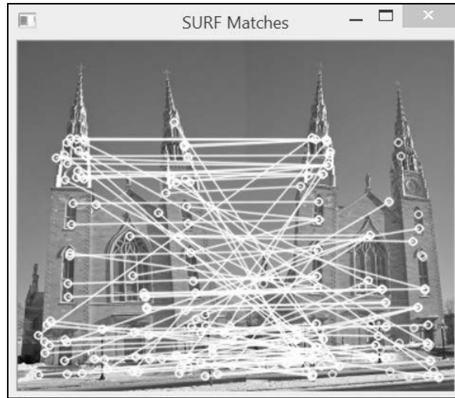
现在可以用这些描述子来进行关键点匹配了。与上节完全一样，将第一幅图像的每个特征描述子向量与第二幅图像的全部特征描述子进行比较。把相似度最高的一对（即两个描述子向量之间的距离最短）保留下来，作为最佳匹配项。对第一幅图像的每个特征，重复上述步骤。这个过程已经在OpenCV的cv::BFMatcher类中实现，使用很方便，因此我们没必要重新实现前面构建的两个循环。类的用法如下：

```
// 构造匹配器
cv::BFMatcher matcher(cv::NORM_L2);
// 匹配两幅图像的描述子
std::vector<cv::DMatch> matches;
matcher.match(descriptors1,descriptors2, matches);
```

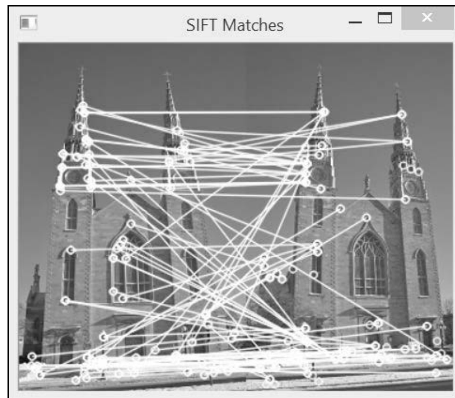
cv::DescriptorMatcher类为不同的匹配策略定义了通用的接口，cv::BFMatcher是它

的子类。返回的结果是一个`cv::DMatch`实例的向量。

采用SURF的Hessian阈值，第一幅图像得到90个关键点，第二幅得到80个关键点。这种brute-force方法（穷举法）将进行90次匹配运算。跟上节一样，使用`cv::drawMatches`类得到如下的图像：

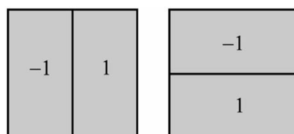


可以看到，有些匹配项正确地连接了左侧的点和右侧对应的点。也有些匹配项是错误的，部分原因是建筑物的对称性导致无法明确地匹配。对于SIFT，采用同样数量的关键点，得到匹配结果如下：



9.3.2　实现原理

好的特征描述子不受照明和视角微小变动的影响，也不受图像中噪声的影响。因此它们通常基于局部强度值的差值。SURF描述子正是如此，它在关键点周围局部地应用下面的简易内核：



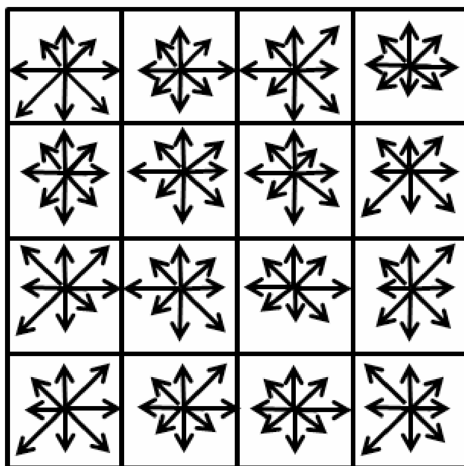
第一个内核度量水平方向的局部强度值差值（标为dx），第二个内核度量垂直方向的差值（标为dy）。用于提取描述子向量的邻域尺寸，通常定为特征值缩放因子的20倍（即20σ）。然后把这个正方形区域划分成更小的4 × 4子区域。对于每个子区域，在5 × 5等分的位置上（用尺寸为2σ的内核）上计算内核反馈值（dx和dy）。用下面的方法累加这些反馈值，为每个子区域提取4个描述子值：

$$\left[\sum \mathrm{d}x \quad \sum \mathrm{d}y \quad \sum|\mathrm{d}x| \quad \sum|\mathrm{d}y|\right]$$

由于子区域的数量是4 × 4 = 16个，因此描述子值的总数为64个。为了赋予邻近像素（即靠近关键点的值）更高的权重，用一个以关键点为中心的高斯算子对内核反馈值进行加权计算（用$\sigma = 3.3$）。

dx和dy反馈值也用于估算特征的方向。在半径为6σ的圆形邻域内计算这些值（内核尺寸为4σ），该邻域的位置用间隔σ进行分片。在指定的方向上，累计某个角度间隔（π/3）内的反馈值，向量最长的方向就定义为主方向。

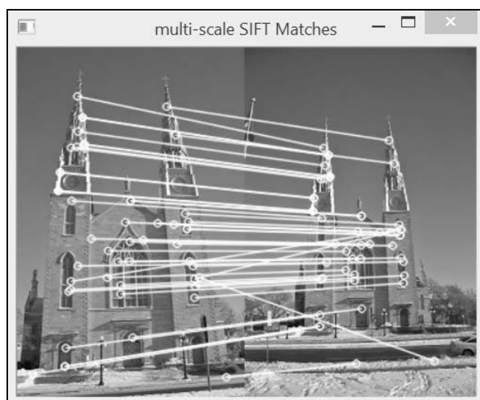
SIFT描述子包含的内容更多，它采用图像梯度而不是单纯的强度差值。它也将关键点周围的正方形邻域分割成4 × 4的子区域（也可以使用8 × 8或2 × 2的子区域）。每个区域内部建立一个梯度方向直方图。这些方向被分隔进8个箱子，每个梯度方向数值的递增量与梯度幅值成正比。下面的图片描述了这个过程，每个星形箭头代表一个局部的梯度方向直方图：



9

这里有16个直方图，每个直方图包含8个连接在一起的箱子，它们形成了一个128维的描述子。对于SURF而言，梯度值是用一个以关键点为中心的高斯滤波器加权计算得到的，用以降低邻域边界上梯度方向的突然变化对描述子的影响。为了使差值度量更加一致，最终的描述子会进行归一化处理。

使用SURF和SIFT的特征和描述子可以进行尺度无关的匹配。下面的例子展示了对两幅不同尺度的图像做SURF匹配的结果（这里显示了50个最佳的匹配项）：



9.3.3　扩展阅读

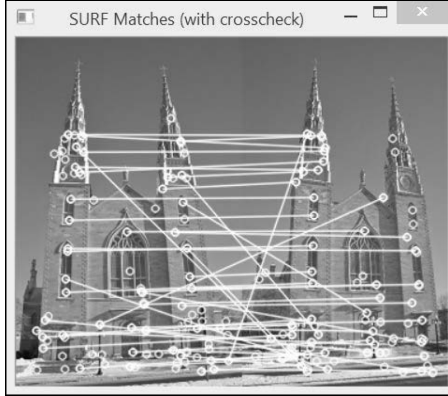
用任何算法得到的匹配结果都含有相当多的错误匹配项。有一些策略可以提高匹配的质量。这里介绍其中两种。

1. 交叉检查匹配项

有一种简单的方法可以验证得到的匹配项，即重新进行同一个匹配过程。第二次匹配时，将第二幅图像的每个关键点逐个与第一幅图像的全部关键点进行比较。只有在两个方向都匹配了同一对关键点（即两个关键点互为最佳匹配），才认为是一个有效的匹配项。函数`cv::BFMatcher`提供了一个选项来使用这个策略。把有关标志设置为`true`，函数就会对匹配进行双向的交叉检查：

```
// 构造带交叉检查的匹配器
cv::BFMatcher matcher2(cv::NORM_L2, // 度量差距
                       true);        // 交叉检查标志
```

改进后的匹配结果如下图所示（用SURF的情形）：



2. 比率检验法

我们已经注意到，正是场景物体中的重复元素导致了匹配结果不可靠，这是因为在匹配视觉上，类似的结构时会产生歧义。这种情况下，一个关键点会与多个关键点很好地匹配。因为选中错误匹配的可能性很大，所以比较可取的做法是排除此类匹配。

要使用这种策略，需要为每个关键点找到两个最佳的匹配项。可以用`cv::DescriptorMatcher`类的`knnMatch`方法实现这个功能。我们只需要两个最佳匹配项，因此指定$k = 2$。

```
// 为每个关键点找出两个最佳匹配项
  std::vector<std::vector<cv::DMatch>> matches2;
  matcher.knnMatch(descriptors1,descriptors2, matches2,
               2); // 找出k个最佳匹配项
```

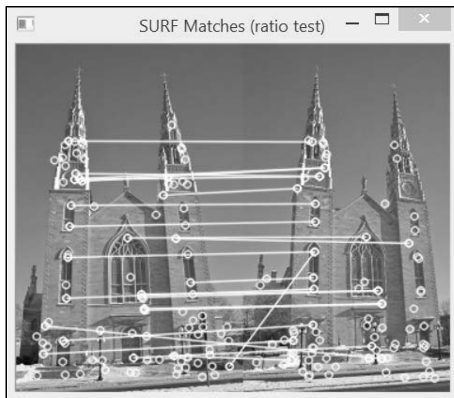
下一步是排除与第二个匹配项非常接近的全部最佳匹配项。因为`knnMatch`生成了一个`std::vector`类型（此向量的长度为k）的`std::vector`类，这一步的具体做法是循环遍历每个关键点匹配项，然后执行比率检验法（(ratio test)，如果两个最佳匹配项相等，那么比率为1）。代码如下：

```
// 执行比率检验法
double ratio= 0.85;
std::vector<std::vector<cv::DMatch>>::iterator it;
for (it= matches2.begin(); it!= matches2.end(); ++it) {

   // 第一个最佳匹配项/第二个最佳匹配项
   if ((*it)[0].distance/(*it)[1].distance < ratio) {
     // 这个匹配项可以接受
     matches.push_back((*it)[0]);
   }
}
// matches是新的匹配项集合
```

原始匹配项集合的90对已减少为现在的23对。正确匹配项的比例已经很高：

9

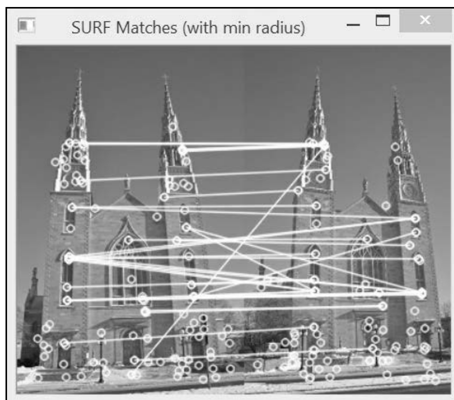


3. 匹配差值的阈值化

还有一种更加简单的策略，就是把描述子之间的差值太大的匹配项排除。实现此功能的是cv::DescriptorMatcher类的radiusMatch方法：

```
// 指定范围的匹配
float maxDist= 0.4;
std::vector<std::vector<cv::DMatch>> matches2;
matcher.radiusMatch(descriptors1, descriptors2, matches2,
                    maxDist); // 两个描述子之间的最大允许差值
```

因为这个方法会保留所有差值小于指定阈值的匹配项，它得到的结果仍是std::vector类型的std::vector类。这说明对于一个关键点，在另一幅图像上可能有多个匹配点。另一方面，其他关键点将会没有匹配项（对应的内部类std::vector的长度为0）。这样，匹配项的数量从原来的90对减少到37对，如下图所示：



当然了，这些策略是可以组合起来使用的，以提升匹配效果。

9.3.4 参阅

- 8.4节介绍了相关的SURF和SIFT特征检测器，并提供了更多的参考资料。
- 10.4节解释了如何利用图像和场景几何形状来获得质量更高的匹配项。
- E. Vincent和R. Laganière发表在*Machine, Graphics and Vision* 2001年第237页至第260页的"Matching feature points in stereo pairs: A comparative study of some matching strategies"描述了其他简单的匹配策略，可用于提高匹配的质量。

9.4 用二值特征描述关键点

上一节我们学习了如何用提取自图像强度值梯度的、丰富的描述子来描述关键点。这些描述子是浮点数类型的向量，大小为64或128，有时甚至更大。这导致对它们的操作所耗资源特别大。为了减少内存使用，降低计算量，最近人们引入了二值描述子的概念。这里的难点在于，既要容易计算，又要在场景和视角变化时保持鲁棒性。本节介绍其中的几种二值描述子。重点讲解ORB和BRISK描述子，第8章介绍了与它们相关的特征点检测器。

9.4.1 如何实现

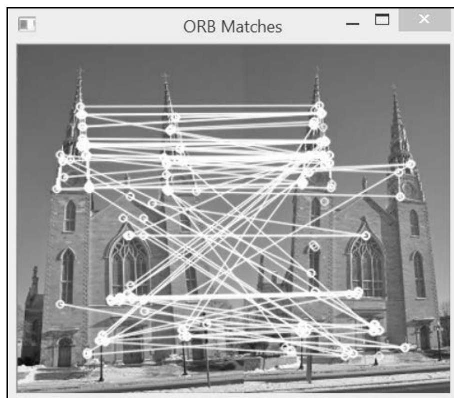
OpenCV的检测器和描述子模块是在通用性很强的接口上构建的，得益于此，ORB等二值描述子的用法与SURF、SIFT等没有什么区别。基于特征的图像匹配的整个过程如下所示：

```
// 定义关键点向量
std::vector<cv::KeyPoint> keypoints1, keypoints2;
// 构造ORB特征检测器对象
cv::Ptr<cv::FeatureDetector> detector =
  new cv::ORB(100); // 检测大约100个ORB特征点
// 检测ORB特征
detector->detect(image1,keypoints1);
detector->detect(image2,keypoints2);
// ORB包含了检测器和描述子提取器
cv::Ptr<cv::DescriptorExtractor> descriptor = detector;
// 提取描述子
cv::Mat descriptors1, descriptors2;
descriptor->compute(image1,keypoints1,descriptors1);
descriptor->compute(image2,keypoints2,descriptors2);
// 构造匹配器
cv::BFMatcher matcher(
     cv::NORM_HAMMING); // 对二值描述子，永远使用hamming规范
// 匹配两个图像描述子
std::vector<cv::DMatch> matches;
matcher.match(descriptors1,descriptors2, matches);
```

唯一的区别是使用了**Hamming**规范（`cv::NORM_HAMMING`标志），它通过统计不一致的位数，计算两个二值描述子的差值。在很多处理器上，这可以通过一个异或运算加上简单的位数统计来

实现，并且效率很高。

下图显示了匹配的结果：



BRISK是另一个常见的二值特征检测器（描述子），用它也能得到类似的结果。这时要调用新的`cv::BRISK(40)`来创建`cv::DescriptorExtractor`实例。和上一章一样，它的第一个参数是一个阈值，用以控制被检测特征点的数量。

9.4.2　实现原理

ORB算法在多个尺度下检测特征点，这些特征点含有方向。基于这些特征点，ORB描述子通过简单比较强度值，提取出每个关键点的表征。实际上ORB就是在BRIEF描述子的基础上构建的（前面介绍过BRIEF描述子）。然后在关键点周围的邻域内随机选取一对像素点，创建一个二值描述子。比较这两个像素点的强度值，如果第一个点的强度值较大，就把对应描述子的位（bit）设为1，否则就设为0。对一批随机像素点对进行上述处理，就产生一个由若干位（bit）组成的描述子。通常采用128到512位（成对地测试）。

这就是ORB采用的模式。接下来，就要判断哪些像素点对可用于构建描述子。事实上，虽然像素点对是随机选取的，一旦被选中就要进行同样的二值测试，并构建全部关键点的描述子，以确保结果的一致性。直觉告诉我们，选择合适的像素点对，可以使描述子具有更大的独特性。并且每个关键点的方向是已经确定的，如果根据方向对该关键点进行调整（即使用相对于关键点方向的坐标），就会导致强度值模式分布的偏差。考虑到这些因素，并根据实验验证，ORB选出了变化幅值较高、相关性极小的256对像素点。也就是说，针对各种关键点，这些选用的二值测试项等于`0`或`1`的几率是均等的，并且它们之间的依赖性是最小的。

`cv::ORB`构造函数除了包括控制特征检测过程的参数，还有两个与描述子有关的参数。一个表示包含像素点对的图像块的尺寸（默认值为31 × 31）。另一个用来测试3个或4个像素点为一组

的点集，而不是默认的像素点对。强烈建议采用默认设置。

BRISK描述子的情况也非常类似，它的基础也是成对地比较强度值。但有两点不同。首先，它不是从31×31的邻域中随机选取像素，而是从一系列等间距的同心圆（由60个点组成）的采样模式中选取。第二，这些采样点的强度值都经过高斯平滑处理，处理中使用的σ值与该像素到圆心的距离成正比。BRISK从这些点中选取512对。

9.4.3　扩展阅读

此外还有一些二值描述子，有兴趣的读者可以查看有关科学文献。这里我们介绍另一种可以在OpenCV中使用的描述子。

FREAK

FREAK全称为Fast Retina Keypoint（快速视网膜关键点）。FREAK也是一种二值描述子，但没有对应的检测器。它可以应用于所有已检测到的关键点，例如SIFT、SURF或ORB。

与BRISK一样，FREAK描述子也基于用同心圆定义的采样模式。但为了设计描述子，设计者们用人类的眼睛做类比。他们发现，随着与中央凹距离的增加，视网膜上的神经节细胞密度会减小。因此他们用43个像素点来构建采样模式，中心点附近的像素密度比其他地方要高得多。为了获得它的强度值，每个像素都用高斯内核进行滤波，当与中心点的距离增加时，内核的尺寸也随之增大。

根据经验，可以采用ORB中使用的类似策略，标识出需要执行的成对比较项。通过对几千个关键点的分析，可得到具有最高变化幅值和最低相关性的二值测试项，最终为512对。

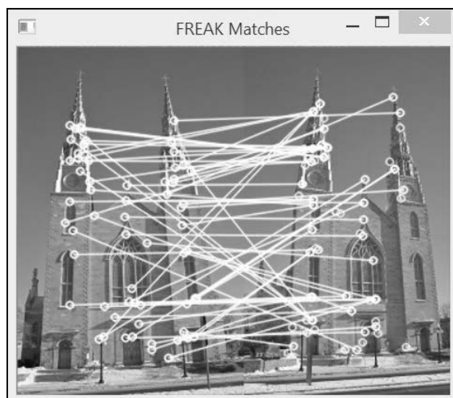
FREAK还引入了阶梯式比较描述子的概念。具体做法是，首先执行表示较粗略的信息的前面128位（用较大的高斯内核，在外围进行测试）。只有对比的描述子通过了第一步测试，后面的测试才能进行。

用ORB算法检测到关键点后，只需用下面的方法创建`cv::DescriptorExtractor`实例即可提取出FREAK描述子：

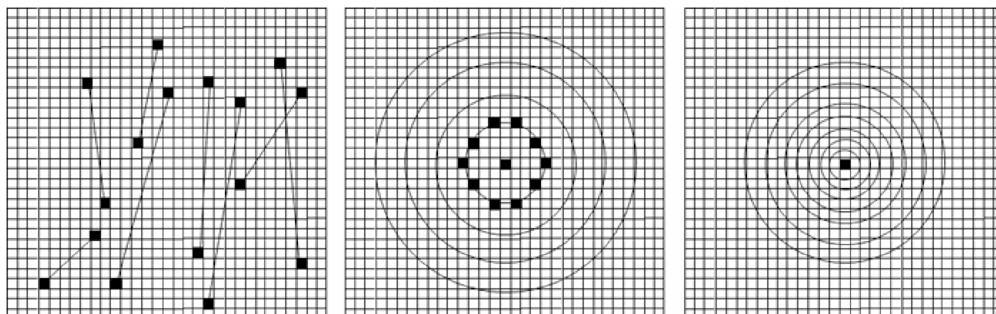
```
cv::Ptr<cv::DescriptorExtractor> descriptor =
   new cv::FREAK(); // 用FREAK描述
```

匹配结果如下：

9



本节的三个描述子所使用的采样模式，见下面的示意图：



第一个方块是ORB描述子，像素点对是在正方形网格中随机选取的。每个像素点对用线条连接起来，表示比较两个像素强度值的概率测试。这里只显示了8对，ORB默认使用256对。中间的方块是BRISK采样模式。在圆形上均匀地采样像素点（显然，这里只显示了第一个圆的点）。第三个方块是FREAK的对数极坐标的采样网格。BRISK的采样点是均匀分布的，而FREAK在近中心点的地方密度更高。例如，BRISK的外围圆圈上有20个点，而FREAK的外围圆圈上只有6个点。

9.4.4　参阅

- 8.5节介绍了相关的BRISK和ORB特征检测器，并提供了更多的参考资料。
- E. M. Calonder、V. Lepetit、M. Ozuysal、T. Trzcinski、C. Strecha和P. Fua发表在*IEEE Transactions on Pattern Analysis and Machine Intelligence*（2012年）的“BRIEF: Computing a Local Binary Descriptor Very Fast”介绍了BRIEF特征描述子，引入二值描述子的概念。
- A.Alahi、R. Ortiz和P. Vandergheynst发表在*IEEE Conference on Computer Vision and Pattern Recognition*（2012年）的“FREAK: Fast Retina Keypoint article”介绍了FREAK特征描述子。

第 10 章

估算图像之间的投影关系

本章包括以下内容：

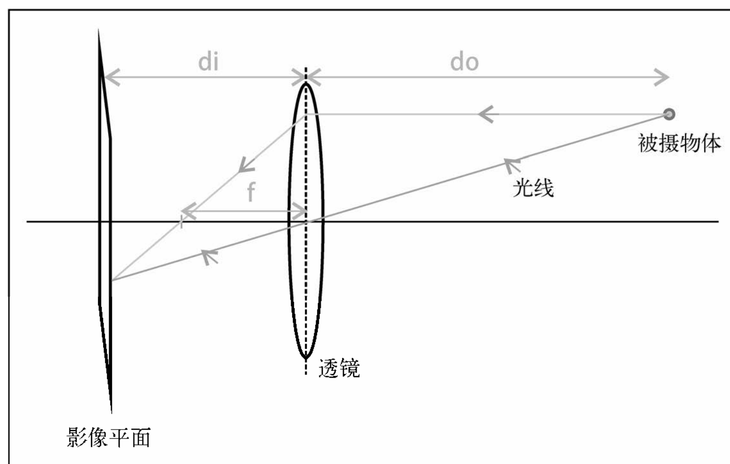
- 相机校准；
- 计算图像对的基础矩阵；
- 用RANSAC算法匹配图像；
- 计算两幅图像之间的单应矩阵。

10.1 简介

通常图像是由数码相机拍摄得到的，它通过透镜投射光线，在图像传感器上捕获场景。图像由三维场景在二维平面上的投影形成，这表明场景和它的图像之间以及同一场景的不同图像之间都有着重要的关联。投影几何学是用数学术语描述和区分成像过程的工具。本章将介绍几种多视图图像中基本的投影关系，并解释如何在计算机视觉编程中应用。你将学会如何通过投影约束使匹配更加精确，如何用双视图关联方法将多幅图像组合成全景图。在开始正文前，我们先探讨与场景投影和成像过程有关的一些基本概念。

成像过程

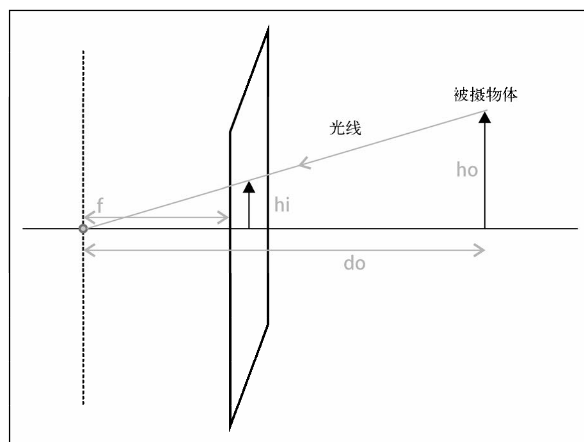
自从照相术发明以来，生成图像的基本过程就没有变过。光线从被摄景象发出并穿过前置孔径，被相机捕获，捕获到的光线触发相机后面的影像平面（或图像传感器）。此外，透镜的使用使来自不同场景元素的光线得以集中。下面的示意图展现了成像过程：



这里的do是透镜到被摄物体的距离，di是透镜到影像平面的距离，f是透镜的焦距。这些数据的关系称为薄镜公式：

$$\frac{1}{f}=\frac{1}{do}+\frac{1}{di}$$

在计算机视觉中，有多种方法可以简化这个相机模型。第一种方法是考虑到相机的口径极小，因而可将镜头的影响忽略不计。从理论上讲，这并不会改变图像的外观。（但是这么做了之后，我们就会创建无限景深的图像而忽略了聚焦效应。）因此，在这种情况下只需考虑中心的光线。第二种方法，因为大多数情况下满足条件`do>>di`，我们可以假定影像平面处于焦点位置。最后一种方法，根据几何学我们可发现影像平面上的图像是反转的。只要把影像平面放在镜头前面，就能得到跟原来几乎一样却不反转的图像。从物理学上看，这显然不可行。但是从数学的角度看，结果完全是等效的。这种简化模型通常称为针孔照相机模型，如下图所示：



根据这个模型和相似三角形的定理，我们可以很容易地推导出表示被摄物体与它的图像的关系的基本投影方程：

$$hi = f\frac{ho}{do}$$

物体（高度为ho）对应的图像大小（hi）与它到相机的距离（do）成反比，这是很自然的规律。通常在相机几何结构已知的情况下，这个关系决定了三维场景的点在影像平面上投影的位置。

10.2　相机校准

根据本章的简介部分，我们知道针孔模型下相机的基本参数，是它的焦距和影像平面的大小（它决定了相机的*视野*）。另外，因为我们处理的是数字图像，所以图像的像素数量（*分辨率*）是相机的另一个重要属性。最后，为了能计算图像上场景点的位置（用像素坐标表示），我们还需要额外的信息。考虑一条从角点出发、与影像平面垂直的直线，我们需要知道这条直线在哪个像素的位置穿透影像平面。这个点称为*像主点*。在逻辑上，我们可以假定像主点位于影像平面的中心位置，但是在实际中这个点会偏离几个像素的距离，具体取决于该相机的制造精度。

相机校准就是设置相机各种参数的过程。当然也可以使用相机厂家提供的技术参数，但是对于某些任务（例如三维重建），这些技术参数是不够精确的。相机校准的过程，就是用相机拍摄特定的图案并分析得到的图像，然后在优化过程中确定最佳的参数值。这是一个复杂的过程，但OpenCV的校准函数已经让它变得很容易。

10.2.1　如何实现

相机校准的基本原理是，确定场景中一系列点的三维坐标并拍摄这个场景，然后观测这些点在图像上投影的位置。有了足够多的三维点和图像上对应的二维点，就可以根据投影方程推断出准确的相机参数。显然，为了得到精确的结果，就要观测尽可能多的点。一种方法是对一个包含大量三维点的场景取像。但是在实际操作中，这种做法几乎是不可行的。更实用的做法是针对一部分三维的点，从不同的视角拍摄多个照片。这种方法相对比较简单，但是它除了需要计算相机本身的参数，还需要计算每个相机视图的位置。

OpenCV推荐使用国际象棋棋盘的图案生成用于校准的三维场景点的集合。这个图案在每个方块的角点位置创建场景点，并且由于图案是平面的，因此我们可以假设棋盘位于Z=0且X和Y的坐标轴与网格对齐的位置。这样，校准时就只需从不同的视角拍摄棋盘图案。下面是一个6 × 4的图案：



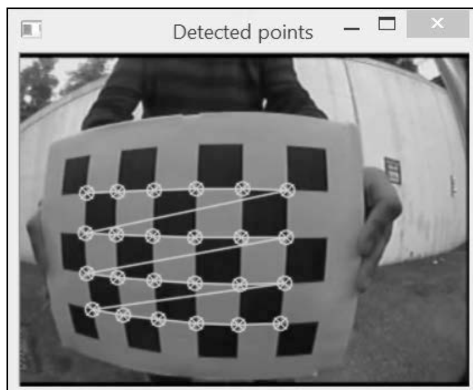
好在OpenCV有一个自动检测棋盘图案中角点的函数。只需要提供一幅图像和棋盘尺寸（水平和垂直方向内部角点的数量），函数就会返回图像中所有棋盘角点的位置。如果无法找到图案，函数返回`false`：

```
// 输出图像角点的向量
std::vector<cv::Point2f> imageCorners;
// 棋盘内部角点的数量
cv::Size boardSize(6,4);
// 获得棋盘角点
bool found = cv::findChessboardCorners(image,
                                        boardSize, imageCorners);
```

输出参数`imageCorners`将存储检测到的内部角点的像素坐标。注意，这个函数还可以使用其他参数来调节算法，这里不讨论。此外还有一个专门的函数来画出棋盘图像上的角点，用线条依次连接起来：

```
//画出角点
cv::drawChessboardCorners(image,
                boardSize, imageCorners,
                found); // 找到的角点
```

得到如下图像:



连接角点的线条的次序，就是角点在向量中存储的次序。在校准前需要指定相关的三维点。指定这些点时可自由选择单位（例如厘米或英寸），不过最简单的办法是指定方块的边长为一个单位。这样第一个点的坐标就是(0, 0, 0)（假设棋盘的纵深坐标为$Z=0$），第二个点的坐标是(1, 0, 0)，依此类推，最后一个点的坐标是(5, 3, 0)。这个图案共有24个点，要进行精确的校准，这些点是远远不够的。为了得到更多的点，需要对同一个校准图案，从不同的视角拍摄更多的照片。可以在相机前移动图案，也可以在棋盘周围移动相机。从数学的角度看，这两种方法是完全等效的。OpenCV的校准函数假定由棋盘图案确定坐标系，并计算相机相对于坐标系的旋转量和平移量。

我们把校准过程封装在`CameraCalibrator`类中。类的属性有：

```
class CameraCalibrator {

    // 输入点：
    // 世界坐标系中的点
    std::vector<std::vector<cv::Point3f>> objectPoints;
    // 点的位置（以像素为单位）
    std::vector<std::vector<cv::Point2f>> imagePoints;
    // 输出矩阵
    cv::Mat cameraMatrix;
    cv::Mat distCoeffs;
    // 指定校准方式的标志
    int flag;
```

注意，输入的场景和像素点的向量其实是`Point`实例组成的`std::vector`。每个向量元素也是一个向量，表示一个视角的点集。这里我们采用增加校准点的方法，指定一个以一批棋盘图像的文件名为输入对象的向量：

```
// 打开棋盘图像，并提取角点
int CameraCalibrator::addChessboardPoints(
         const std::vector<std::string>& filelist,
         cv::Size & boardSize) {

   // 棋盘上的角点
   std::vector<cv::Point2f> imageCorners;
   std::vector<cv::Point3f> objectCorners;

   // 场景中的三维点：
   // 在棋盘坐标系中，初始化棋盘中的角点
   // 角点的三维坐标(X,Y,Z)= (i,j,0)
   for (int i=0; i<boardSize.height; i++) {
      for (int j=0; j<boardSize.width; j++) {
         objectCorners.push_back(cv::Point3f(i, j, 0.0f));
      }
    }

    // 图像上的二维点：
    cv::Mat image; // 用于存储棋盘图像
    int successes = 0;
```

10

```
        // 处理所有视角
        for (int i=0; i<filelist.size(); i++) {
            // 打开图像
            image = cv::imread(filelist[i],0);
            // 取得棋盘中的角点
            bool found = cv::findChessboardCorners(
                            image, boardSize, imageCorners);
            // 取得角点上的子像素级精度
            cv::cornerSubPix(image, imageCorners,
                      cv::Size(5,5),
                      cv::Size(-1,-1),
             cv::TermCriteria(cv::TermCriteria::MAX_ITER +
                              cv::TermCriteria::EPS,
                      30,      // 最大迭代次数
                      0.1));  // 最小精度

            //如果棋盘是完好的，就把它加入结果
            if (imageCorners.size() == boardSize.area()) {
                // 加入从同一个视角得到的图像和场景点
                addPoints(imageCorners, objectCorners);
                successes++;
            }
        }
       return successes;
    }
```

第一个循环中输入棋盘的三维坐标，然后通过函数cv::findChessboardCorners获得图像中对应的点。图像中所有可能的视角都会执行该过程。并且使用cv::cornerSubPix函数可以使图像上点的位置更精确，正如函数名所示，它能以子像素级精度定位图像中的点。用cv::TermCriteria对象指定终止的条件，它定义了最大迭代次数和最小子像素级坐标精度。只要这两个条件中有一个满足，这个角点细化过程就会结束。

在成功地检测到一批棋盘角点后，用自定义的addPoints方法把这些点加入图像和场景点的向量。处理完足够数量的棋盘图像后（这时就有了大量的三维场景点/二维图像点的对应关系），就可以开始计算校准参数：

```
    // 校准相机
    // 返回重投影误差
    double CameraCalibrator::calibrate(cv::Size &imageSize)
    {
       // 输出旋转量和平移量
        std::vector<cv::Mat> rvecs, tvecs;

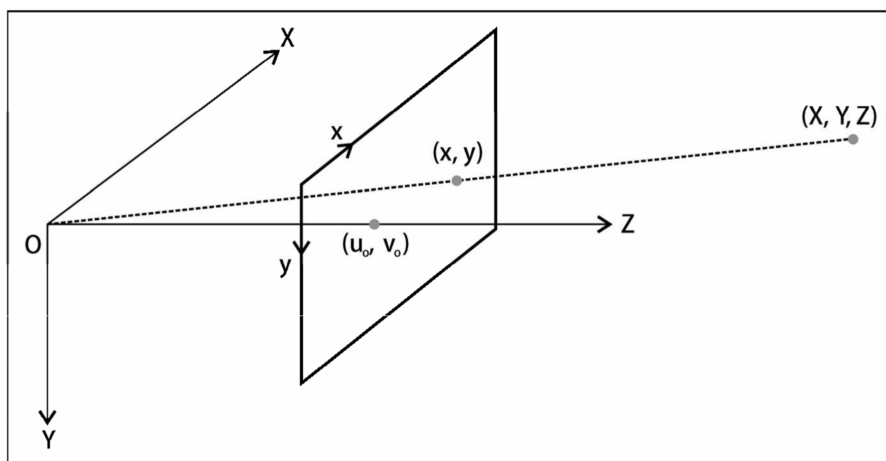
       // 开始校准
       return
         calibrateCamera(objectPoints, // 三维点
                   imagePoints,  // 图像点
                   imageSize,    // 图像尺寸
                   cameraMatrix, // 输出相机矩阵
                   distCoeffs,   // 输出畸变矩阵
```

```
                    rvecs, tvecs, // Rs、Ts
                    flag);        // 设置选项
}
```

根据经验，10~20个棋盘图像就足够了，但是必须在不同的深度，从不同的视角拍摄。这个函数的两个重要输出对象是相机矩阵和畸变参数。后面会详细介绍。

10.2.2 实现原理

为了解释校准的结果，我们重新看一下简介部分的针孔相机模型示意图。尤其是要论证三维点(X, Y, Z)和对应的图像点(x, y)之间的关系，其中相机中的图像点用像素坐标表示。我们在示意图的投影中心加上坐标系，如下图所示：



习惯上我们把左上角作为图像的原点，为了保证坐标系与之保持一致，这里的Y坐标轴是向下的。前面我们学过，(X, Y, Z)点会投影到图像平面的$(f_X/Z, f_Y/Z)$位置。现在为了转换成像素坐标，需要把二维图像的位置分别除以像素宽度（p_x）和高度（p_y）。注意，以世界单位（一般用毫米）表示的焦距除以p_x后，就得到以像素单位表示的焦距（水平方向）。同样的，$f_y=f/p_y$是以像素为单位的垂直方向的焦距。因此完整的投影方程为：

$$x = \frac{f_x X}{Z} + u_0$$

$$y = \frac{f_y Y}{Z} + v_0$$

前面讲过，（u_0, v_0）是像主点，将其添加到结果，以实现将原点移到图像的左上角的目的。也可以通过引入齐次坐标，以矩阵形式改写这些方程。在齐次坐标中，用三维向量表示一个二维

点，用四维向量表示一个三维点（额外的坐标只是一个任意的缩放因子，即$\boldsymbol{S}$，从齐次坐标的三维向量中提取二维坐标时要去掉这个额外的坐标）。

改写后的投影方程如下：

$$\boldsymbol{S}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

第二个矩阵只是一个投影矩阵。第一个矩阵包含了相机的全部参数，称作相机的内部参数。这个3×3矩阵是`cv::calibrateCamera`函数输出矩阵中的一个。此外还有一个`cv::calibrationMatrixValues`函数，它以校准矩阵的形式返回内部参数的值。

一般而言，如果坐标系不是位于相机投影的中心，我们就需要加上一个旋转向量（3 × 3的矩阵）和一个平移向量（3 × 1的矩阵）。这两个矩阵描述了刚体变换。为了变换回相机的坐标系，必须在三维点上应用刚体变换。于是可以把投影方程写成更通用的形式：

$$\boldsymbol{S}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} r1 & r2 & r3 & t1 \\ r4 & r5 & r6 & t2 \\ r7 & r8 & r9 & t3 \end{bmatrix}\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

我们知道在这个校准例子中，坐标系统是位于棋盘上的。因此必须对每个视图计算刚体变换（由一个旋转量和一个平移量组成，旋转量用矩阵入口$r1$到$r9$表示，平移量用$t1$、$t2$和$t3$表示）。它们在`cv::calibrateCamera`函数的输出参数列表中。旋转和平移部分通常称为校准的外部参数，并且对于每个视图它们都各不相同。对于指定的相机/镜头，内部参数是不变的。本例基于20个棋盘图像进行校准，得到相机的内部参数，它们是f_x=167、f_y=178、u_0=156和v_0=119。这些值是`cv::calibrateCamera`函数通过优化过程获得的，该优化过程的目的是找到内部和外部参数，实现图像点的预定义位置和实际位置之间差距的最小化。其中预定义位置根据三维场景点的投影计算得到，实际位置通过图像观测得到。校准过程中所有点的这种差距之和，称为重投影误差。

现在我们来看畸变参数。到现在为止，我们一直认为在针孔相机模型下，镜头的影响是可以忽略的。但这仅限于镜头在抓取图像时不会产生严重视觉畸变的情况。但如果使用劣质镜头或者镜头的焦距太短，情况就不同了。也许你已经注意到了，本例图像中的棋盘图案已经明显变形了——矩形棋盘的边线已经扭曲。另外，从图像中心移开时，畸变会变得更加严重。这是超广角镜头产生的典型畸变，称为径向畸变。一般数码相机的镜头通常不会产生这么高的畸变，

但是对于本例使用的镜头，显然是不能忽略畸变的。

采用合适的畸变模型，可以对这些变形进行补偿。其原理是用一系列数学公式表示因镜头产生的畸变。公式在建立后可以进行还原，以恢复图像中可见的畸变。幸好，在校准阶段可以获得纠正畸变所需的准确变换参数以及其他相机参数。完成这个步骤后，用刚校准的相机拍摄的所有图像都不会有畸变。因此，我们在校准类中增加了一个额外的方法：

```
// 去除图像中的畸变（校准后）
cv::Mat CameraCalibrator::remap(const cv::Mat &image) {

   cv::Mat undistorted;

   if (mustInitUndistort) { // 每个校准过程调用一次

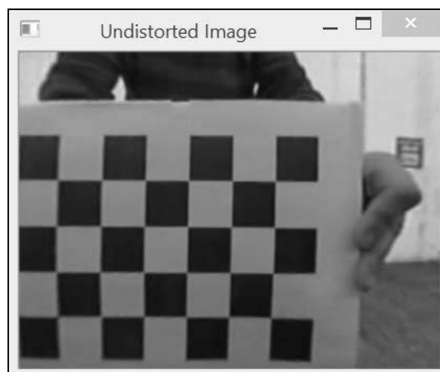
    cv::initUndistortRectifyMap(
      cameraMatrix,  // 计算得到的相机矩阵
      distCoeffs,    // 计算得到的畸变矩阵
      cv::Mat(),     // 可选矫正项（无）
      cv::Mat(),     // 生成无畸变的相机矩阵
      image.size(),  // 无畸变图像的尺寸
      CV_32FC1,      // 输出图片的类型
      map1, map2);   // x和y映射功能

    mustInitUndistort= false;
   }

   // 应用映射功能
   cv::remap(image, undistorted, map1, map2,
      cv::INTER_LINEAR); // 插值类型

   return undistorted;
}
```

运行这段代码后得到如下图像：



可以发现，在畸变被纠正后就得到了有规则的透视图。

为了纠正畸变，OpenCV使用一个多项式函数，把图像点移动到未畸变的位置。默认使用5个系数，也可以采用8个系的模型。得到系数后就可以计算两个cv::Mat类型的映射函数（分别用于x坐标和y坐标），把有畸变的图像上的像素点映射到未畸变的新位置。函数cv::initUndistortRectifyMap计算映射值，函数cv::remap把输入图像的点映射到新图像上。注意，因为是非线性变换，输入图像中的部分像素映射后会超出输出图像的边界。可以扩大输出图像的尺寸以弥补像素丢失，但是这样的话就需要填充在输入图像中没有值的输出像素（它们将显示为黑色像素）。

10.2.3　扩展阅读

在相机校准过程中可以使用更多的选项。

1. 用已知的内部参数进行校准

如果可以准确估算相机的内部参数，那么将这些参数输入函数cv::calibrateCamera将十分有利。它们可作为优化处理过程的初始值。要实现这一步操作，你只需添加CV_CALIB_USE_INTRINSIC_GUESS标志，并在校准矩阵参数中输入这些值。也可以把像主点强制设为某个固定的值（CV_CALIB_FIX_PRINCIPAL_POINT），通常假定它就是中心点的像素。还可以把焦距fx和fy强制设成某个固定的比例（CV_CALIB_FIX_RATIO）。在此情况下，假设像素集是一个正方形。

2. 使用圆形组成的网格进行校准

除了通常使用的棋盘图案，OpenCV还可以使用由圆形组成的网格进行相机校准。这时就用圆心作为校准点。对应的函数与前面定位棋盘角点的函数非常类似：

```
cv::Size boardSize(7,7);
std::vector<cv::Point2f> centers;
bool found = cv:: findCirclesGrid(
                    image, boardSize, centers);
```

10.2.4　参阅

- 10.5节将检查特殊情况下的投影方程。
- Z. Zhang发表在*IEEE Transactions on Pattern Analysis and Machine Intelligence* 2000年第22卷第11期的“A flexible new technique for camera calibration”是解决相机校准问题的经典论文。

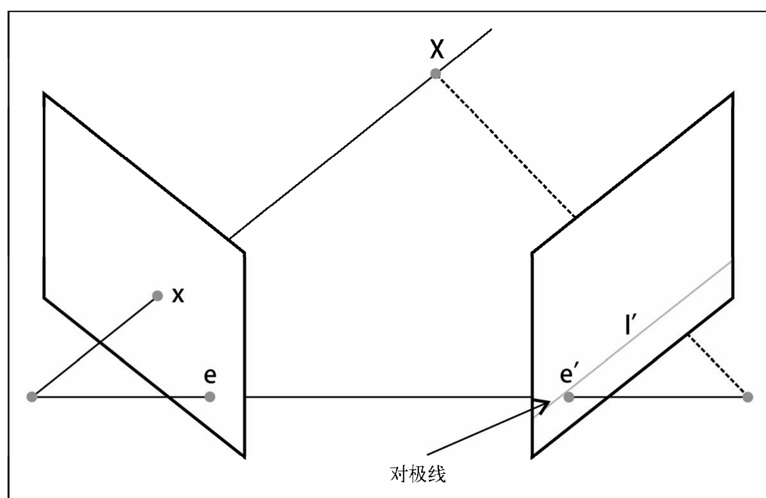
10.3　计算图像对的基础矩阵

上一节介绍了如何恢复单个相机的投影方程。本节将探讨同一场景的两幅图像之间的投影关系。可以移动单个相机，从两个视角拍摄两幅照片。也可以使用两个相机，分别对同一个场景拍

摄照片。如果这两个相机被刚性基线分割，我们就称之为立体视觉。

10.3.1 准备工作

现在来看两个相机观察同一个场景点的情况，如下图所示：



我们知道，沿着三维点X和相机中心点之间的连线，可在图像上找到对应的点x。反过来，在三维空间中，与图像平面上的位置x对应的场景点可以位于线条上的任何位置。这说明如果要根据图像中的一个点找到另一幅图像中对应的点，就需要在第二个图像平面上沿着这条线的投影搜索。这条虚线称为点x的对极线。它规定了两个对应点必须满足的基本条件，即对于一个点，在另一视图中与它匹配的点必须位于它的对极线上，并且对极线的准确方向取决于两个相机的相对位置。事实上，对极线的结构决定了双视图系统的几何形状。

从这个双视图系统的几何形状中还能发现一个现象，即所有的对极线都通过同一个点。这个点对应着一个相机中心点在另一个相机上的投影。这个特殊的点称为极点。

图像上的点和它的对极线之间的关系，在数学上可以用下面的3 × 3矩阵表示：

$$\begin{bmatrix} l_1' \\ l_2' \\ l_3' \end{bmatrix} = \boldsymbol{F} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

在投影几何学中，可以用三维向量表示二维直线。它就是一些二维点的集合(x',y')，满足公式$\boldsymbol{I}_1'x'+\boldsymbol{I}_2'y'+\boldsymbol{I}_3'=0$（上标符号表示这条线属于第二幅图像）。因此，矩阵$\boldsymbol{F}$（称为基础矩阵）的作用就是把一个视图上的二维图像点映射到另一个视图上的对极线上。

10.3.2　如何实现

如果在两个图像之间含有一定数量的已知匹配点，就可以利用方程组来计算图像对的基础矩阵。这样的匹配项至少要有7对。为了说明基础矩阵的计算过程和图像对的用法，我们可人为地选择7对较好的匹配项。用OpenCV函数`cv::findFundamentalMat`计算基础矩阵时，将使用这些匹配项，如下图所示：



如果已有的图像点是`cv::keypoint`类型的（例如第8章的关键点检测结果），为了在`cv::findFundamentalMat`中使用，需要先转换成`cv::Point2f`类型。为此可用下面的OpenCV函数：

```
// 把关键点转换成Point2f
std::vector<cv::Point2f> selPoints1, selPoints2;
std::vector<int> pointIndexes1, pointIndexes2;
cv::KeyPoint::convert(keypoints1,selPoints1,pointIndexes1);
cv::KeyPoint::convert(keypoints2,selPoints2,pointIndexes2);
```

向量`selPoints1`和`selPoints2`包含了两幅图像中相对应的点。关键点实例是`keypoints1`和`keypoints2`。向量`pointIndexes1`和`pointIndexes2`包含被转换关键点的序号。然后调用`cv::findFundamentalMat`函数，方法如下：

```
// 用7对匹配项计算基础矩阵
cv::Mat fundamental= cv::findFundamentalMat(
   selPoints1,    // 第一个图像中的7个点
   selPoints2,    // 第二个图像中的7个点
   CV_FM_7POINT); // 7个点的方法
```

要直观地验证这个基础矩阵的效果，可以选取一些点，画出它们的对极线。OpenCV中有一个函数可计算指定点集的对极线。计算出对极线后，可用函数`cv::line`画出它们。下面的代码片段完成了这两个步骤（即在右侧图像中计算和画出来自左侧图像的对极线）：

```
// 在右侧图像上画出对极线的左侧点
```

```
std::vector<cv::Vec3f> lines1;
cv::computeCorrespondEpilines(
   selPoints1,  // 图像点
   1,           // 在图像1中 (也可以是2)
   fundamental, // 基础矩阵
   lines1);     // 对极线的向量

// 遍历全部对极线
for (vector<cv::Vec3f>::const_iterator it= lines1.begin();
    it!=lines1.end(); ++it) {
       // 画出第一列和最后一列之间的线条
       cv::line(image2,
         cv::Point(0,-(*it)[2]/(*it)[1]),
         cv::Point(image2.cols,-((*it)[2]+
                   (*it)[0]*image2.cols)/(*it)[1]),
                   cv::Scalar(255,255,255));
}
```

结果如下图所示：



极点位于所有对极线的交叉点，并且它是另一个相机中心点的投影。上面的图片中能看到极点。对极线的交叉点通常在图像边界的外面。在我们的例子中，如果两个图像是同时拍摄的，极点就位于能看到第一个相机的位置。用7对匹配项计算基础矩阵，得到的结果可能是非常不稳定的。事实上，如果用一对匹配项代替另一对，就会产生一组明显不同的对极线。

10.3.3 实现原理

前面我们解释过，对于图像上的一个点，可以根据基础矩阵得到一个线条的方程式，并在这个线条上找到另一个视图中与之对应的点。假设点p（用齐次坐标表示）的对应点为p'，两个视图间的基础矩阵为$\boldsymbol{F}$。由于p'位于对极线$\boldsymbol{F}p$上，因此可得到如下公式：

$$p'^{T} \boldsymbol{F} p = 0$$

这个公式表示两个对应点之间的关系，称为极线约束。利用这个公式，就可以根据已知的匹配项计算矩阵的入口。因为基础矩阵的入口数量取决于尺度因子，所以需要计算的入口只有8个（第9个可以直接设置为1）。每个匹配项产生一个方程式。有了8个已知匹配项，就可以通过对线性方程组的求解，计算出整个矩阵。可以通过在函数`cv::findFundamentalMat`中采用`CV_FM_8POINT`标志来完成这个过程。注意这时可以（最好）输入8个以上的匹配项。在均方意义下，可以解出线性方程组的超定系统。

计算基础矩阵时可以使用一个附加的约束条件。从数学上看，基础矩阵把一个二维点映射到一个一维的直线束上（即相交于同一个点的直线）。所有对极线都穿过同一个点（极点），对矩阵产生了一个约束条件。这个约束条件把计算基础矩阵所需的匹配次数减少到7次。不过这种情况下，方程组变为非线性的，最多会有三种结果（这时函数`cv::findFundamentalMat`将返回9 × 3的基础矩阵，包含三个3 × 3矩阵）。在OpenCV中，可通过使用`CV_FM_7POINT`标志，采用7次匹配方案计算基础矩阵。

最后要提到的是，如果想要精确地计算基础矩阵，对匹配项的选择是很重要的。一般来说，匹配项要在整幅图像中均匀分布，并包含场景中不同深度的点，否则结果就会不稳定或变差。尤其是如果选择了位于同一平面的场景点，基础矩阵（本例中）就会变差。

10.3.4　参阅

- R. Hartley和A. Zisserman的*Multiple View Geometry in Computer Vision*（剑桥大学出版社，2004年）是有关计算机视觉投影几何学最完整的参考书。
- 下一节将解释如何用更多的匹配项稳定地计算基础矩阵。
- 如果匹配点位于同一平面或是纯旋转的结果，就无法计算基础矩阵，10.5节将解释其中的原因。

10.4　用 RANSAC（随机抽样一致性）算法匹配图像

两个相机拍摄同一个场景时，会在不同的视角下看到相同的元素。我们已经在上一章学过了特征点匹配问题，本节我们来重新探讨这个问题，并学习如何开发两个视图之间的极线约束，使图像特征的匹配更加可靠。

我们遵循的原则很简单：在匹配两幅图像的特征点时，只接受位于对极线上的匹配项。而要判断这个条件，必须先知道基础矩阵，但是计算基础矩阵前需要有优质的匹配项。这看起来像是“先有鸡还是先有蛋”的问题。本节提出了一种解决方案，可以同时计算基础矩阵和一批优质的匹配项。

10.4.1 如何实现

我们的目的是计算两个视图间的基础矩阵和优质匹配项，因此，所有已发现的特征点的匹配度都要用上节的极线约束来验证。我们为此创建了一个类，封装了鲁棒的匹配过程的各个步骤：

```
class RobustMatcher {
  private:
    // 特征点检测器对象的指针
    cv::Ptr<cv::FeatureDetector> detector;
    // 特征描述子提取器对象的指针
    cv::Ptr<cv::DescriptorExtractor> extractor;
    int normType;
    float ratio; // 第一个和第二个NN之间的最大比率
    bool refineF; // 如果等于true，则会优化基础矩阵
    double distance; // 到极点的最小距离
    double confidence; // 可信度（概率）

  public:
    RobustMatcher(std::string detectorName, // 用名称表示
                  std::string descriptorName)
      : normType(cv::NORM_L2), ratio(0.8f),
          refineF(true), confidence(0.98), distance(3.0) {

      // 用名称构造
      if (detectorName.length()>0) {

      detector= cv::FeatureDetector::create(detectorName);
      extractor= cv::DescriptorExtractor::
                          create(descriptorName);
      }
    }
```

注意这里是如何使用接口`cv::FeatureDetector`和`cv::DescriptorExtractor`的`create`方法的，这样开发者就可以根据名称选择合适的`create`方法。也可以用获取方法`setFeatureDetector`和`setDescriptorExtractor`来指定`create`方法。

核心方法是`match`，它返回匹配项、被检测的关键点和计算得到的基础矩阵。方法内部有四个独立的步骤（在代码中用注释进行了明显的划分），我们将详细探讨：

```
// 用RANSAC算法匹配特征点
// 返回基础矩阵和输出的匹配项
cv::Mat match(cv::Mat& image1, cv::Mat& image2, // 输入图像
    std::vector<cv::DMatch>& matches,           // 输出匹配项
    std::vector<cv::KeyPoint>& keypoints1,      // 输出关键点
    std::vector<cv::KeyPoint>& keypoints2) {

    // 1. 检测特征点
    detector->detect(image1,keypoints1);
    detector->detect(image2,keypoints2);

    // 2. 提取特征描述子
```

```
    cv::Mat descriptors1, descriptors2;
    extractor->compute(image1,keypoints1,descriptors1);
    extractor->compute(image2,keypoints2,descriptors2);

    // 3. 匹配两幅图像描述子
    //   (用于部分检测方法)

    // 构造匹配类的实例(带交叉检查)
    cv::BFMatcher matcher(normType, //差距衡量
                          true); // 交叉检查标志
    // 匹配描述子
    std::vector<cv::DMatch> outputMatches;
    matcher.match(descriptors1,descriptors2,outputMatches);

    // 4. 用RANSAC算法验证匹配项
    cv::Mat fundaental= ransacTest(outputMatches,
                            keypoints1, keypoints2, matches);

    // 返回基础矩阵
    return fundamental;
  }
```

前面两个步骤只是检测特征点并计算它们的描述子。接下来和上一章一样使用cv::BFMatcher类执行特征匹配。为提高匹配质量而使用了交叉检查标志。

第四个步骤是本节介绍的新概念。它包含了一个额外的过滤测试，在本例中表现为使用基础矩阵来排除不符合极线约束的匹配项。这个测试基于RANSAC算法，即使匹配项中仍有局外项，该算法仍然可以计算基础矩阵（下一节会解释这个方法）：

```
// 用RANSAC算法标识优质的匹配项
// 返回基础矩阵和输出匹配项
cv::Mat ransacTest(const std::vector<cv::DMatch>& matches,
                   const std::vector<cv::KeyPoint>& keypoints1,
                   const std::vector<cv::KeyPoint>& keypoints2,
                   std::vector<cv::DMatch>& outMatches) {

// 关键点转换成Point2f类型
  std::vector<cv::Point2f> points1, points2;
  for (std::vector<cv::DMatch>::const_iterator it=
  matches.begin(); it!= matches.end(); ++it) {

       // 取得左侧关键点的位置
       points1.push_back(keypoints1[it->queryIdx].pt);
       // 取得右侧关键点的位置
       points2.push_back(keypoints2[it->trainIdx].pt);
    }

  // 用RANSAC算法计算基础矩阵
  std::vector<uchar> inliers(points1.size(),0);
  cv::Mat fundamental= cv::findFundamentalMat(
      points1,points2, // 匹配的点
      inliers,      // 匹配状态(局内项或局外项)
```

```
        CV_FM_RANSAC, // RANSAC方法
        distance,     // 到对极线的距离
        confidence);  // 可信度

    // 提取存留的（局内）匹配项
    std::vector<uchar>::const_iterator itIn= inliers.begin();
    std::vector<cv::DMatch>::const_iterator itM= matches.begin();

    // 遍历全部匹配项
    for ( ;itIn!= inliers.end(); ++itIn, ++itM) {
      if (*itIn) { // 有效的匹配项
        outMatches.push_back(*itM);
      }
    }
    return fundamental;
  }
```

这段代码比较长，因为在计算基础矩阵之前要把关键点转换成`cv::Point2f`类型。利用这个类，鲁棒地匹配图像对就非常方便了，调用方法如下：

```
// 准备匹配器（用默认参数）
RobustMatcher rmatcher("SURF"); // 这里使用SURF特征
// 匹配两幅图像
std::vector<cv::DMatch> matches;
std::vector<cv::KeyPoint> keypoints1, keypoints2;
cv::Mat fundamental= rmatcher.match(image1,image2,
                            matches, keypoints1, keypoints2);
```

结果得到62个匹配项，如下图所示：



有趣的是，除了几个偶尔会落在基础矩阵对应的对极线上的错误匹配项，几乎所有的匹配项都是正确的。

10.4.2　实现原理

在上一节我们学过，可以根据一些特征点匹配项计算图像对的基础矩阵。为了确保结果准确，所采用匹配项必须都是优质的。但是在实际情况中，通过比较被检测特征点的描述子得到的匹配项是无法保证全部准确的。正因为如此，人们引入了基于RANSAC（RANdom SAmpling Consensus）策略的基础矩阵计算方法。

RANSAC算法旨在根据一个可能包含大量局外项的数据集，估算一个特定的数学实体。其原理是从数据集中随机选取一些数据点，并仅用这些数据点进行估算。选取的数据点数量，应该是估算数学实体所需的最小数量。对于基础矩阵，最小数量是8个匹配对（实际上只需要7个，但是8个点的线性算法速度较快）。用这8个随机匹配对估算基础矩阵后，对剩下的全部匹配项进行测试，验证其是否满足根据这个矩阵得到的极线约束。标识出所有满足极线约束的匹配项（即特征点与对极线距离很近的匹配项）。这些匹配项就组成了基础矩阵的*支撑集*。

RANSAC算法背后的核心思想是：支撑集越大，所计算矩阵正确的可能性就越大。反之，如果一个（或多个）随机选取的匹配项是错误的，那么计算得到的基础矩阵也是错误的，并且它的支撑集肯定会很小。反复执行这个过程，最后留下支撑集最大的矩阵，作为最佳结果。

因此我们的任务就是随机选取8个匹配项，重复多次，最后得到8个合适的匹配项，能产生很大的支撑集。如果在整个数据集中，错误匹配项的数量不同，那么选取到8个正确匹配项的可能性也各不相同。但是我们知道，选取的次数越多，在这些选项中至少有一组优质匹配的可能性就越大。更准确地说，假设匹配项中局内项（优质匹配项）的比例是`w%`，那么选取8个优质匹配项的概率就是`w%`。因此，在一次选取中至少包含一个错误匹配项的概率是`(1-w)`。如果选取的次数是`k`，其中有一次只包含优质匹配项的选取的概率是`1-(1-w)k`。这就是置信概率，标为`c`。要得到正确的基础矩阵，需要至少一个优质匹配集，因此我们希望这个概率尽可能的大。所以在运行RANSAC算法时，我们需要确定得到特定可信度等级所需的选取次数`k`。

在使用含有`CV_FM_RANSAC`标志的函数`cv::findFundamentalMat`时，会提供两个附加参数。第一个参数是可信度等级，它决定了执行迭代的次数（默认值是`0.99`）。第二个参数是点到对极线的最大距离，小于这个距离的点被视为局内点。如果匹配对中有一个点到对极线的距离超过这个值，就视这个匹配对为局外项。这个函数返回字符值的`std::vector`，表示对应的输入匹配项被标记为局外项（`0`）或局内项（`1`）。

匹配集中优质匹配项越多，RANSAC算法得到正确基础矩阵的概率就越大。因此我们在匹配特征点时使用了交叉检查过滤器。还可以使用上节介绍的比率测试，进一步提高最终匹配集的质量。这是一个互相权衡的问题，要考虑这三点：计算复杂度、最终匹配项数量、要得到仅包含准确匹配项的匹配集所需的可信度等级。

10.4.3 扩展阅读

本节介绍的鲁棒匹配过程，即利用拥有最大支架的8个被选匹配项以及该支撑集所包含的匹配项计算基础矩阵，然后得到基础矩阵的估算值。利用这个信息，有两种方法可以改进这些结果。

1. 改进基础矩阵

现在我们已经有了高质量的匹配项，最好的办法就是在最后用全部匹配项重新估算基础矩阵。我们已经注意到，有一种线性的8点算法可以估算这个矩阵。因此可以得到一个超定方程组，求得最小二乘法形式的基础矩阵。可以在ransacTest函数的后面添加这个步骤：

```
if (refineF) {
  // 用全部认可的匹配项重新计算基础矩阵

  // 把关键点转换成Point2f类型
  points1.clear();
  points2.clear();

  for (std::vector<cv::DMatch>::
              const_iterator it= outMatches.begin();
     it!= outMatches.end(); ++it) {

     // 取得左侧关键点的位置
     points1.push_back(keypoints1[it->queryIdx].pt);

     // 取得右侧关键点的位置
     points2.push_back(keypoints2[it->trainIdx].pt);
  }

  // 根据全部认可的匹配项，计算8个点的基础矩阵
  fundamental= cv::findFundamentalMat(
    points1,points2, // 匹配点
    CV_FM_8POINT);   // 8个点的方法，用奇异值分解（SVD）求解
}
```

实际上cv::findFundamentalMat函数可以通过奇异值分解求解线性方程组的方式，接受8个以上的匹配项。

2. 改进匹配项

我们知道在双视图系统中，每个点肯定位于与它对应的点的对极线上。这就是基础矩阵所表示的极线约束。因此，如果已经有了很准确的基础矩阵，就可以利用极线约束来更正得到的匹配项，具体做法是将强制匹配项置于它们的对极线上。使用OpenCV函数cv::correctMatches可以很方便地实现这个功能：


```
std::vector<cv::Point2f> newPoints1, newPoints2;
// 改进匹配项
correctMatches(fundamental,            // 基础矩阵
              points1, points2,        // 原始位置
```

```
                    newPoints1, newPoints2); // 新位置
```

这个函数修改每个对应点的位置，从而在最小化累积（平方）位移时能满足极线约束。

10.5　计算两幅图像之间的单应矩阵

10.2节介绍了用匹配项计算图像对的基础矩阵的方法。在投影几何学中，还有一种非常实用的数学实体。这个实体可以用多视图影像计算，是一个具有特殊性质的矩阵。

10.5.1　准备工作

这次我们仍考虑三维点和它在相机中的影像之间的投影关系，10.1节曾介绍过。我们知道这个公式的本质是利用相机内部参数和相机的位置（用旋转分量和平移分量表示），建立三维点和它的影像之间的关联关系。仔细研究这个公式，会发现有两种特殊情况需要我们注意。第一种情况，即同一场景的两个视图之间差别只有纯旋转。这时外部矩阵的第四列全部变为0（即没有平移量）：

$$\boldsymbol{S}\begin{bmatrix}x\\y\\1\end{bmatrix}=\begin{bmatrix}f_x & 0 & u_0\\0 & f_y & v_0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}r1 & r2 & r3 & 0\\r4 & r5 & r6 & 0\\r7 & r8 & r9 & 0\end{bmatrix}\begin{bmatrix}X\\Y\\Z\\1\end{bmatrix}$$

于是在这种特殊情况下，投影关系就变为3×3的矩阵。如果拍摄目标是一个平面，也会出现类似的有趣现象。这种特殊情况下，我们可以假设平面上的点都位于Z=0的位置，这样仍能保持通用性。最终得到下面的公式：

$$\boldsymbol{S}\begin{bmatrix}x\\y\\1\end{bmatrix}=\begin{bmatrix}f_x & 0 & u_0\\0 & f_y & v_0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}r1 & r2 & r3 & t1\\r4 & r5 & r6 & t2\\r7 & r8 & r9 & t3\end{bmatrix}\begin{bmatrix}X\\Y\\0\\1\end{bmatrix}$$

场景点中值为0的坐标会消除掉投影矩阵的第三列，从而又变成一个3 × 3的矩阵。这种特殊矩阵称为单应矩阵（homography），表示在特殊情况下（这里是纯旋转或平面目标），点和它的影像之间是线性关系，格式如下：

$$\begin{bmatrix}sx'\\sy'\\s\end{bmatrix}=\boldsymbol{H}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

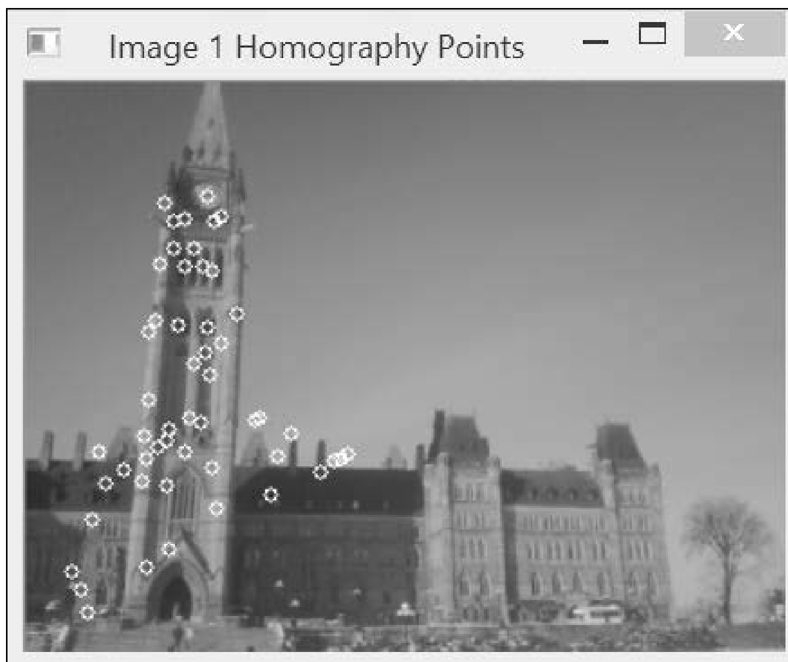
其中$\boldsymbol{H}$是一个3×3矩阵。这个关系式包含了一个尺度因子，用s表示。计算得到这个矩阵后，一个视图中的所有点都可以根据这个关系式转换到另一个视图。需要注意的是，在使用单应矩阵关系式后，基础矩阵就没有意义了。

10.5.2　如何实现

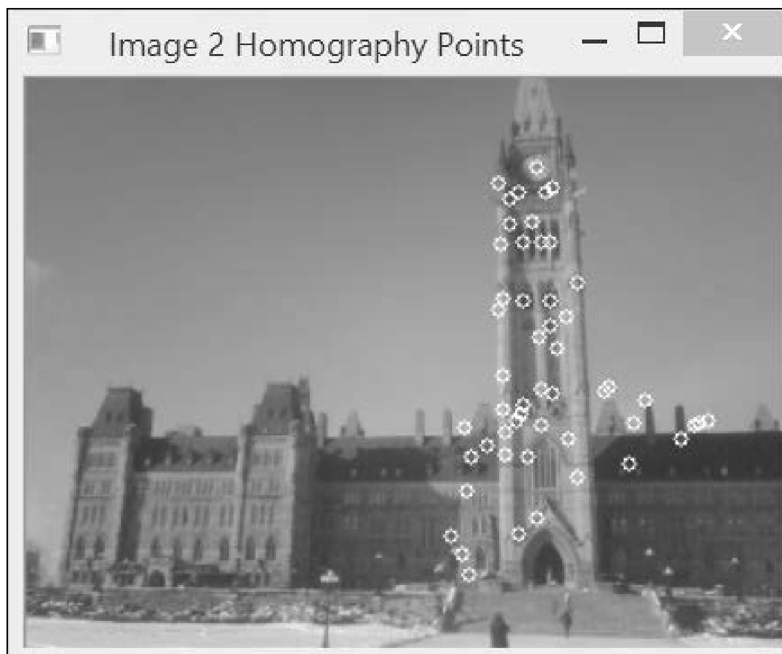
假设我们有两幅图像，它们的差别在于纯旋转量。如果你一边转动自己一边对建筑物或风景拍照，就会出现这种情况。只要拍摄者与目标的距离足够远，平移量就可以忽略不计。可以选取特征点，使用`cv::BFMatcher`函数匹配这两幅图像。跟上节一样，接下来我们对此应用RANSAC算法，这次包含基于匹配集（显然有大量的局外项）估算单应矩阵的步骤。该步骤通过`cv::findHomography`函数实现，和`cv::findFundamentalMat`函数很相似：

```
// 找到图像1和图像2之间的单应矩阵
std::vector<uchar> inliers(points1.size(),0);
cv::Mat homography= cv::findHomography(
  points1, points2, // 对应的点
  inliers,    // 输出的局内匹配项
  CV_RANSAC, // RANSAC方法
  1.);        // 到重复投影点的最大距离
```

这里有单应矩阵（而不是基础矩阵），是因为两幅图像的差距为纯旋转量。下面展示了这两幅图像，同时展示了用`inliers`参数识别出的局内关键点。参见下图：



第二幅图像为：



前面画出了依据单应矩阵得到的局内点，用的是下面的循环代码：

```
// 画出局内点
std::vector<cv::Point2f>::const_iterator itPts=
                                         points1.begin();
std::vector<uchar>::const_iterator itIn= inliers.begin();
while (itPts!=points1.end()) {

   // 在每个局内点的位置画一个圆
   if (*itIn)
       cv::circle(image1,*itPts,3,
                 cv::Scalar(255,255,255));
   ++itPts;
   ++itIn;
}
```

单应矩阵是一个3×3的可逆矩阵。因此，在计算单应矩阵后，就可以把一幅图像的点转移到另一幅图像。实际上，图像中的每个像素都可以转移。因此可以把整幅图像迁移到另一幅图像的的视点上。这个过程称为图像拼接，常用于根据多幅图像构建一幅大型全景图。OpenCV中有一个函数能实现这个功能，用法如下：

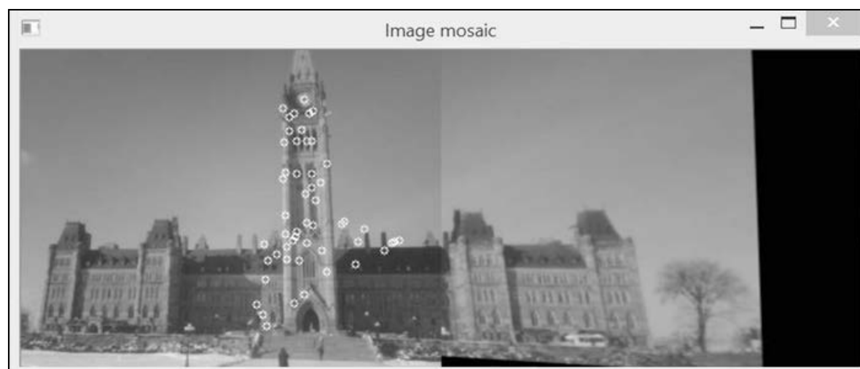
```
// 扭曲图像1到图像2
cv::Mat result;
cv::warpPerspective(image1,  // 输入图像
```

```
        result,                    // 输出图像
        homography,                // 单应矩阵
        cv::Size(2*image1.cols,
                image1.rows)); // 输出图像的尺寸
```

得到新图像后，可以把它附加到另一幅图像上以扩展视角（因为现在两幅图像的视角是相同的）：

```
// 把图像1复制到完整图像的第一个半边
cv::Mat half(result,cv::Rect(0,0,image2.cols,image2.rows));
image2.copyTo(half); // 把image2复制到image1的ROI区域
```

结果如下图所示：



10.5.3 实现原理

如果两个视图通过单应矩阵互相关联，就可以检测一副图像中特定的场景点在另一幅图像中的位置。如果一幅图像中的点对应到另一幅图像后超出了边界范围，这种性质就显得尤其有趣。实际上，由于第二个视图中有一部分场景在第一个视图中是不可见的，因此可以用单应矩阵，通过读取另一幅图像中额外的像素点的颜色值来扩展图像。这样我们就能通过扩充第二幅图像得到新的图像，此时新图像的右侧会添加额外的列。

用函数`cv::findHomography`计算得到的单应矩阵，把第一幅图像的点映射到第二幅图像的点。计算单应矩阵时至少需要四个匹配项，并且它也使用了RANSAC算法。在找到具有最佳支撑集的单应矩阵后，`cv::findHomography`方法就会用全部识别到的局内项对它进行优化。

现在要把图像1的点迁移到图像2，需要做的实际上就是反转单应矩阵。这正是函数`cv::warpPerspective`的默认算法，即利用输入的反转单应矩阵取得输出图像中每个像素的颜色值（这就是第2章中的反向映射）。如果迁移的输出像素超出了输入图像的范围，就把这个像素设置为黑色（`0`）。如果要在转移像素的过程中直接使用单应矩阵，而不是反转矩阵，就可以在函数`cv::warpPerspective`的第五个可选参数中指定`cv::WARP_INVERSE_MAP`标志。

10.5.4 扩展阅读

同一平面的两幅图像之间也存在单应矩阵。我们可以用它来识别图像中的平面物体。

检测图像中的平面目标

假定我们需要检测图像中存在的平面物体。这个物体可能是一张海报、一幅画、一个标牌、一本书的封面（下面的例子），等等。利用本章学到的知识，我们采取的方法是检测这个物体的特征点，然后试着在图像中匹配这些特征点。然后用鲁棒匹配方案来验证这些匹配项，这个方案与上一节的类似，但这次是基于单应矩阵的。

定义一个TargetMatcher类，它与RobustMatcher非常相似：

```
class TargetMatcher {

  private:

    // 特征点检测器对象的指针
    cv::Ptr<cv::FeatureDetector> detector;
    // 特征描述子提取器对象的指针
    cv::Ptr<cv::DescriptorExtractor> extractor;
    cv::Mat target; // 目标图像
    int normType;
    double distance; // 最小重投影误差
```

这里只是增加了一个target属性，表示被匹配的平面物体的参考图像。这些匹配方法与RobustMatcher类中的方法是一样的。区别在于它们在ransacTest方法中包含了cv::findHomography，而不是cv::findFundamentalMat。另外还增加了一个方法来初始化匹配目标，并找到目标的位置：

```
    // 检测图像中定义的平面物体
    // 返回单应矩阵
    // 被检测目标的4个角点，以及匹配项和关键点
    cv::Mat detectTarget(const cv::Mat& image,
      // 目标角点的位置（顺时针方向）
      std::vector<cv::Point2f>& detectedCorners,
      std::vector<cv::DMatch>& matches,
      std::vector<cv::KeyPoint>& keypoints1,
      std::vector<cv::KeyPoint>& keypoints2) {

      // 找到目标和图像之间的RANSAC单应矩阵
      cv::Mat homography= match(target,image,matches,
                                keypoints1, keypoints2);

      // 目标角点
      std::vector<cv::Point2f> corners;
      corners.push_back(cv::Point2f(0,0));
      corners.push_back(cv::Point2f(target.cols-1,0));
      corners.push_back(cv::Point2f(target.cols-1,target.rows-1));
```

```
    corners.push_back(cv::Point2f(0,target.rows-1));

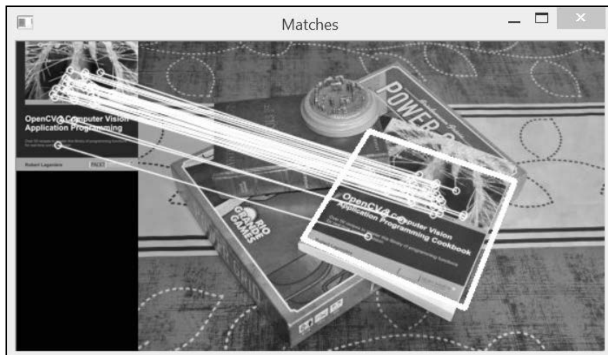
    // 重新投射目标角点
    cv::perspectiveTransform(corners,detectedCorners,
                                      homography);
    return homography;
  }
```

在用匹配方法得到单应矩阵后，我们就能定义目标的四个角点（即参考图像的四个角点）。然后用cv::perspectiveTransform函数把这些角点转移到图像中。这个函数只是用单应矩阵放大输入向量中的每个点。这样就得到了这些点在另一幅图像中的坐标。然后用下面的方法进行目标匹配：

```
// 准备匹配器
TargetMatcher tmatcher("FAST","FREAK");
tmatcher.setNormType(cv::NORM_HAMMING);

// 定义输出数据
std::vector<cv::DMatch> matches;
std::vector<cv::KeyPoint> keypoints1, keypoints2;
std::vector<cv::Point2f> corners;
// 参考图像
tmatcher.setTarget(target);
// 匹配图像与目标
tmatcher.detectTarget(image,corners,matches,
                              keypoints1,keypoints2);
// 画出图像中的目标角点
cv::Point pt= cv::Point(corners[0]);
cv::line(image,cv::Point(corners[0]),cv::Point(corners[1]),
                cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[1]),cv::Point(corners[2]),
                cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[2]),cv::Point(corners[3]),
                cv::Scalar(255,255,255),3);
cv::line(image,cv::Point(corners[3]),cv::Point(corners[0]),
                cv::Scalar(255,255,255),3);
```

用cv::drawMatches函数显示结果，如下图所示：



我们还可以用单应矩阵来修改平面物体的透视图。例如，有几幅从不同视角拍摄建筑物正面的照片，可以计算这些图像之间的单应矩阵，并且用本节的方法扭曲并组合图像，从而构建出建筑物正面的大型全景图。计算单应矩阵时至少需要两个视图之间的四个匹配点。可以使用`cv::getPerspectiveTransform`函数，从四个对应点进行这样的变换。

10.5.5　参阅

- 2.8节讨论了反向映射的概念。
- M.Brown和D.Lowe发表在*International Journal of Computer Vision*（2007年1月，第74期）的“Automatic panoramic image stitching using invariant features”描述了用多幅图像构建全景图的完整方法。

第 11 章

处理视频序列

本章包括以下内容：

- 读取视频序列；
- 处理视频帧；
- 写入视频帧；
- 跟踪视频中的特征点；
- 提取视频中的前景物体。

11.1 简介

视频信号是重要的视觉信息来源。视频由一系列图像构成，这些图像称为帧，帧是以固定的时间间隔获取的（称为帧速率，通常用帧/秒表示），据此可以显示运动中的场景。随着高性能计算机的出现，现在已经能够对视频序列进行高级的视觉分析——被分析的帧速率可以接近甚至超过实际视频的帧速率。本章介绍如何读取、处理和存储视频序列。

我们将看到，如果从视频序列中提取出独立的帧，就可以使用本书介绍的各种图像处理函数。此外我们将学习几种对视频序列做时序分析的算法，为跟踪物体而比较相邻的帧或者为提取前景物体而在时间上累计图像统计数据。

11.2 读取视频序列

要处理视频序列，首先要读取每个帧。OpenCV提供了一个便于使用的框架来提取帧，帧的来源可以是视频文件，也可以是USB或IP摄像机。本节将介绍它的用法。

11.2.1 如何实现

总体来说，要从视频序列读取帧，只需创建一个`cv::VideoCapture`类的实例，然后在一个循环中提取并读取每个视频帧。下面这个基本的`main`函数显示了视频序列中的帧：

```
int main()
{
  // 打开视频文件
  cv::VideoCapture capture("bike.avi");
  // 检查打开是否成功
  if (!capture.isOpened())
    return 1;

  // 取得帧速率
  double rate= capture.get(CV_CAP_PROP_FPS);

  bool stop(false);
  cv::Mat frame; // 当前视频帧
  cv::namedWindow("Extracted Frame");

  // 根据帧速率计算帧之间的等待时间，单位ms
  int delay= 1000/rate;

  // 循环遍历视频中的全部帧
  while (!stop) {

    // 读取下一帧（如果有）
    if (!capture.read(frame))
      break;

    cv::imshow("Extracted Frame",frame);

    // 等待一段时间，或者通过按键停止
    if (cv::waitKey(delay)>=0)
      stop= true;
  }

  // 关闭视频文件
  // 不是必需的，因为类的析构函数会调用
  capture.release();
  return 0;
}
```

程序会显示一个播放视频的窗口，如下图所示：



11.2.2 实现原理

只需指定视频文件名即可打开视频，可以在cv::VideoCapture对象的构造函数中指定文件名。如果cv::VideoCapture对象已经创建，也可以使用它的open方法。成功打开视频后（可用isOpened方法验证），就可以开始提取帧。也可使用get方法并采用正确的标志，通过cv::VideoCapture对象查询视频文件的有关信息。前面的例子中，我们用CV_CAP_PROP_FPS标志获得帧速率。因为它是一个通用函数，所以即使有的时候需要其他类型，它也总会返回一个double类型的数值。例如可用下面的方法获得视频文件的总帧数（整数）：

```
long t= static_cast<long>(
              capture.get(CV_CAP_PROP_FRAME_COUNT));
```

要了解能从视频获得的信息类型，请查阅OpenCV文档提供的各种标志。

此外还可以用set方法输入cv::VideoCapture实例的参数。例如可以用CV_CAP_PROP_POS_FRAMES标志，让视频跳转到指定的帧：

```
// 跳转到第100帧
double position= 100.0;
capture.set(CV_CAP_PROP_POS_FRAMES, position);
```

还可以用CV_CAP_PROP_POS_MSEC以毫秒为单位指定位置，或者用CV_CAP_PROP_POS_AVI_RATIO指定视频内部的相对位置（0.0表示视频开始位置，1.0表示结束位置）。如果参数设置成功，函数会返回true。对于一个特定的视频来说，参数能否读取或设置，在很大程度上取决于用来压缩和存储视频序列的编解码器。如果某些参数不能使用，可能就是由编解码器造成的。

在成功地打开视频后，可以像前面的例子那样反复调用read方法，按顺序访问每一帧。也可调用重载的读取运算子，作用完全一样：

```
capture >> frame;
```

还能使用这两个基本方法：

```
capture.grab();
capture.retrieve(frame);
```

注意我们在显示每一帧时所采用的延时方法，使用了cv::waitKey函数。这里采用的延时时长取决于视频的帧频率（假设fps为每秒的帧数，1000/fps就是两个帧之间的毫秒数）。可以通过修改这个数值，让视频慢进或快进。在播放视频时有一点很重要，就是采用的延时时长要保证窗口有足够的时间进行刷新（因为这是一个低优先级的进程，如果CPU太忙就不会刷新）。使用cv::waitKey函数，可以通过按任意键中断这个读取过程。这时，函数会返回按键的ASCII码。注意，如果cv::waitKey函数中指定的延时为0，那么它将永远等待下去，直到用户按下一个键。这种方法非常适用于需要通过逐帧检查以跟踪一个过程的情况。

最后的语句调用release方法关闭视频文件。不过这并不是必需的，因为在cv::VideoCapture的析构函数中也会调用release。

有一点需要特别注意，电脑中必须安装有相关的编解码器，才能打开指定的视频文件，否则cv::VideoCapture将无法对文件进行解码。一般来说，如果用视频播放器（例如Windows Media Player）可以打开该视频文件，那么OpenCV也能读取它。

11.2.3　扩展阅读

还可以连接摄像机和电脑，读取摄像机（例如USB摄像机）捕获的视频流。只需在open函数中指定一个ID（整数），取代原来的文件名。ID为0表示打开默认摄像机。这种情况下就必须用cv::waitKey函数来终止处理过程，因为摄像机视频流的读取过程是不会结束的。

最后，也可以装载Web上的视频。需要提供一个正确的网址，例如：

```
cv::VideoCapture capture("http://www.laganiere.name/bike.avi");
```

11.2.4　参阅

- 11.4节更详细地介绍了视频编解码器。
- 网站http://ffmpeg.org/上有音频/视频读取、记录、转换和成流的完整源码和跨平台解决方案。

11.3　处理视频帧

本节的目标，是针对视频序列中的每一帧应用几个处理函数。我们创建一个自定义类，封装OpenCV的视频捕获框架。此外可以在这个类中指定一个函数，每次提取到新的帧时就会调用该函数。

11.3.1　如何实现

我们要指定一个函数（回调函数），视频序列的每一帧都会调用它。该函数定义为输入一个cv::Mat实例，输出一个处理完毕的帧。因此，框架中有效的回调函数必须遵循以下签名：

```
void processFrame(cv::Mat& img, cv::Mat& out);
```

作为处理函数的例子，我们来看下面这个简单的函数，它的功能是计算输入图像的Canny边缘：

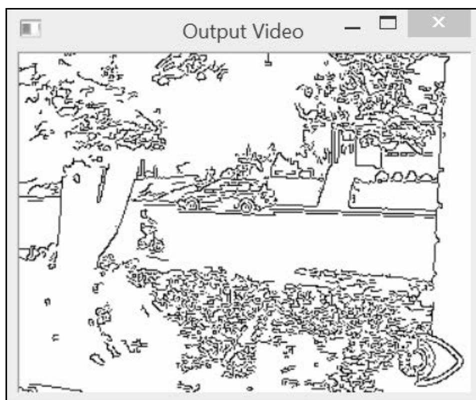
```
void canny(cv::Mat& img, cv::Mat& out) {
  // 转换成灰度图像
  if (img.channels()==3)
    cv::cvtColor(img,out,CV_BGR2GRAY);
```

```
  // 计算Canny边缘
  cv::Canny(out,out,100,200);
  // 反转图像
  cv::threshold(out,out,128,255,cv::THRESH_BINARY_INV);
}
```

自定义类VideoProcessor完整地封装了视频处理任务。使用这个类的程序可以创建类的实例、指定输入图像文件、指定回调函数，然后开始处理。编程时这些步骤都是用这个自定义类实现的，代码如下：

```
// 创建实例
VideoProcessor processor;
// 打开视频文件
processor.setInput("bike.avi");
// 声明显示视频的窗口
processor.displayInput("Current Frame");
processor.displayOutput("Output Frame");
// 用原始帧速率播放视频
processor.setDelay(1000./processor.getFrameRate());
// 设置处理帧的回调函数
processor.setFrameProcessor(canny);
// 开始处理
processor.run();
```

运行这段代码，会在两个窗口中播放输入图像和输出结果，播放速率为原始帧速率（因为用setDelay方法做了延时处理）。如果输入上节用于显示帧的视频，输出窗口如下：



11.3.2 实现原理

跟别的章节一样，我们要创建一个封装视频处理算法通用功能的类。类中包含一些成员变量，用于控制处理视频帧的各种参数：

```
class VideoProcessor {
```

```
private:

// OpenCV视频捕获对象
cv::VideoCapture capture;
// 处理每一帧时都会调用的回调函数
void (*process)(cv::Mat&, cv::Mat&);
// 布尔型变量，表示该回调函数是否会被调用
bool callIt;
// 输入窗口的显示名称
std::string windowNameInput;
// 输出窗口的显示名称
std::string windowNameOutput;
// 帧之间的延时
int delay;
// 已经处理的帧数
long fnumber;
// 达到这个帧数时结束
long frameToStop;
// 结束处理
bool stop;
```

第一个成员变量是cv::VideoCapture对象。第二个属性是函数指针process，它指向一个回调函数。可以用对应的获取方法指定这个函数：

```
// 设置针对每一帧调用的回调函数
void setFrameProcessor(
  void (*frameProcessingCallback)
    cv::Mat&, cv::Mat&)) {

      process= frameProcessingCallback;
}
```

下面的方法打开视频文件：

```
// 设置视频文件的名称
bool setInput(std::string filename) {

  fnumber= 0;
  // 防止已经有资源与VideoCapture实例关联
  capture.release();
  // 打开视频文件
  return capture.open(filename);
}
```

显示经过处理的帧通常会比较有趣。因此我们用两个方法来创建显示窗口：

```
// 用于显示输入的帧
void displayInput(std::string wn) {

  windowNameInput= wn;
  cv::namedWindow(windowNameInput);
}
```

```
// 用于显示处理过的帧
void displayOutput(std::string wn) {

  windowNameOutput= wn;
  cv::namedWindow(windowNameOutput);
}
```

主函数名为run，它包含了提取帧的循环：

```
// 抓取（并处理）序列中的帧
void run() {

  // 当前帧
  cv::Mat frame;
  // 输出帧
  cv::Mat output;

  // 如果没有设置捕获设备
  if (!isOpened())
    return;

  stop= false;

  while (!isStopped()) {

    // 读下一帧（如果有）
    if (!readNextFrame(frame))
      break;

    // 显示输入的帧
    if (windowNameInput.length()!=0)
      cv::imshow(windowNameInput,frame);

    // 调用处理函数
    if (callIt) {

      // 处理帧
      process(frame, output);
      // 递增帧数
      fnumber++;

      } else { // 没有处理
        output= frame;
      }

      // 显示输出的帧
      if (windowNameOutput.length()!=0)
        cv::imshow(windowNameOutput,output);

      // 产生延时
      if (delay>=0 && cv::waitKey(delay)>=0)
        stopIt();

    // 检查是否需要结束
```

```
      if (frameToStop>=0 &&
        getFrameNumber()==frameToStop)
          stopIt();
    }
  }

  // 结束处理
  void stopIt() {

    stop= true;
  }

  // 处理过程是否已经停止?
  bool isStopped() {

    return stop;
  }

  // 捕获设备是否已经打开?
  bool isOpened() {

    capture.isOpened();
  }

  // 设置帧之间的延时,
  // 0表示每一帧都等待,
  // 负数表示不延时
  void setDelay(int d) {

    delay= d;
  }
```

这个方法使用了一个用于读取帧的private方法：

```
  // 取得下一帧,
  // 可以是：视频文件或者摄像机
  bool readNextFrame(cv::Mat& frame) {

    return capture.read(frame);
  }
```

run方法首先调用OpenCV的类cv::VideoCapture的read方法，然后执行一系列操作。但是在执行之前，要先检查该操作是否需要执行。只有指定了输入窗口的名称（用displayInput方法），才会显示输入窗口；只有指定了回调函数（用setFrameProcessor方法），才会运行回调函数；只有定义了输出窗口的名称（用displayOutput），才会显示输出窗口；只有指定了延时（用setDelay），才会执行延时。最后，如果定义了需处理的最大帧数（用stopAtFrameNo），就需要检查当前的帧数。

你或许还希望打开并播放视频文件（不调用回调函数）。所以我们准备了两个方法，以指定是否需要调用回调函数：

```
// 需要调用回调函数process
void callProcess() {

  callIt= true;
}

// 需要调用回调函数process
void dontCallProcess() {

  callIt= false;
}
```

最后，可指定是否需要在处理完一定数量的帧后就结束：

```
void stopAtFrameNo(long frame) {

  frameToStop= frame;
}

// 返回下一帧的编号
long getFrameNumber() {

  // 从捕获设备获取信息
  long fnumber= static_cast<long>(
    capture.get(CV_CAP_PROP_POS_FRAMES));
  return fnumber;
}
```

类中还包含了一些设计方法和获取方法，这些方法基本上只是封装了`cv::VideoCapture`框架的常规方法`set`和`get`。

11.3.3 扩展阅读

使用`VideoProcessor`类有助于视频处理模块的部署。它还可以做几项改进。

1. 处理图像序列

有时输入序列是一批独立存储的图像。简单地改动一下这个类就可以适应这种输入。只需要添加一个成员变量，该变量存储了图像文件名向量和对应的迭代器：

```
// 作为输入对象的图像文件名向量
std::vector<std::string> images;
// 图像向量的迭代器
std::vector<std::string>::const_iterator itImg;
```

用新的`setInput`方法指定需要读取的文件：

```
// 设置输入图像的向量
bool setInput(const std::vector<std::string>& imgs) {
```

```
  fnumber= 0;
  // 防止已经有资源与VideoCapture实例关联
  capture.release();

  // 将这个图像向量作为输入对象
  images= imgs;
  itImg= images.begin();

  return true;
}
```

isOpened方法现在修改成这样：

```
// 捕获设备是否已经打开?
bool isOpened() {

  return capture.isOpened() || !images.empty();
}
```

最后需要修改私有方法readNextFrame，改成根据输入内容选择从视频读取还是从文件名向量读取。判断方法是查看图像文件名向量是否为空，如不为空就表明输入是图像序列。调用setInput并传入视频文件名，将清空该向量：

```
// 取得下一帧
// 可以是：视频文件、摄像机、图像向量
bool readNextFrame(cv::Mat& frame) {

  if (images.size()==0)
    return capture.read(frame);

  else {

      if (itImg != images.end()) {

        frame= cv::imread(*itImg);
        itImg++;
        return frame.data != 0;

      } else

        return false;
  }
}
```

2. 使用帧处理类

在面向对象的编程中，最好使用帧处理类而不是帧处理函数。实际上，在定义视频处理算法时，使用类能提供更大的灵活性。我们可以定义一个接口，在VideoProcessor内部使用的每个类都需要实现该接口：

```
// 处理帧的接口
class FrameProcessor {
```

```
  public:
  // 处理方法
  virtual void process(cv:: Mat &input, cv:: Mat &output)= 0;
};
```

你可在设置方法中为VideoProcessor框架输入一个FrameProcessor实例，把这个实例赋给新增的成员变量frameProcessor，这个成员变量是指向FrameProcessor对象的指针：

```
// 设置实现FrameProcessor接口的实例
void setFrameProcessor(FrameProcessor* frameProcessorPtr)
{

  // 使回调函数失效
  process= 0;
  // 这个就是即将被调用的帧处理接口
  frameProcessor= frameProcessorPtr;
  callProcess();
}
```

在指定帧处理实例后，要让以前设置的帧处理函数失效。如果指定的是一个帧处理函数，也需要让以前设置的实例失效。run方法中的while循环也要做相应的修改：

```
while (!isStopped()) {

  // 读取下一帧（如果有）
  if (!readNextFrame(frame))
    break;

  // 显示输入的帧
  if (windowNameInput.length()!=0)
    cv::imshow(windowNameInput,frame);

  // ** 调用处理函数或方法 **
  if (callIt) {

    // 处理帧
    if (process) // 如果是回调函数
      process(frame, output);
    else if (frameProcessor)
      // 如果是类的接口
      frameProcessor->process(frame,output);
    // 递增帧数
    fnumber++;

  } else {

    output= frame;
  }

   // 显示输出的帧
   if (windowNameOutput.length()!=0)
     cv::imshow(windowNameOutput,output);
```

```
    // 产生延时
    if (delay>=0 && cv::waitKey(delay)>=0)
      stopIt();
    // 检查是否需要结束
    if (frameToStop>=0 && getFrameNumber()==frameToStop)
      stopIt();
}
```

11.3.4　参阅

- 11.5节中有使用FrameProcessor接口类的例子。

11.4　写入视频帧

在前面几节，我们学了如何读取视频文件并提取其帧。本节将介绍如何写入帧并创建视频文件。这样我们就完成了典型的视频处理过程：读取视频流，处理其中的帧，然后在新的视频文件中存储结果。

11.4.1　如何实现

OpenCV用cv::VideoWriter类写视频文件。类的构造函数中可指定文件名、播放所生成视频的帧速率、每个帧的尺寸、是否创建彩色视频：

```
writer.open(outputFile, // 文件名
    codec,          // 所用的编解码器
    framerate,      // 视频的帧速率
    frameSize,      // 帧的尺寸
    isColor);       // 彩色视频?
```

另外，必须指明保存视频数据的方式，即codec参数。本节的最后部分将详细探讨。

打开视频文件后，可以通过反复地调用write方法，在视频文件中加入帧：

```
writer.write(frame); // 在视频文件中加入帧
```

简单地改动上节的VideoProcessor类，就可以增加用cv::VideoWriter类写视频文件的功能。下面是一个简单的程序，包含读视频、处理视频和把结果写入视频文件等功能：

```
// 创建实例
VideoProcessor processor;

// 打开视频文件
processor.setInput("bike.avi");
processor.setFrameProcessor(canny);
processor.setOutput("bikeOut.avi");
// 开始处理
```

```
processor.run();
```

跟上节一样，用户要能选择把帧写入独立的图像。框架中采用的命名规则由前缀和固定位数的数字组成。在存储帧的时候，这个数字会自动递增。为了把输出结果保存到一系列图像中，需要这样修改上面的语句：

```
processor.setOutput("bikeOut", //前缀
".jpg",      // 扩展名
3,           // 数字的位数
0)// 开始序号
```

用这个位数，调用时会创建bikeOut000.jpg、bikeOut001.jpg和bikeOut002.jpg等文件。

11.4.2　实现原理

现在介绍如何修改`VideoProcessor`类，使它能写入视频文件。首先必须添加一个`cv::VideoWriter`类型的成员变量（还有几个其他属性）：

```
class VideoProcessor {

  private:

  ...
  // OpenCV写视频对象
  cv::VideoWriter writer;
  // 输出文件名
  std::string outputFile;
  // 输出图像的当前序号
  int currentIndex;
  // 输出图像文件名中数字的位数
  int digits;
  // 输出图像的扩展名
  std::string extension;
```

用一个额外的方法指定（并打开）输出视频文件：

```
// 设置输出视频文件
// 默认情况下会使用与输入视频相同的参数
bool setOutput(const std::string &filename, int codec=0,
  double framerate=0.0, bool isColor=true) {

  outputFile= filename;
  extension.clear();

  if (framerate==0.0)
    framerate= getFrameRate(); // 与输入相同

  char c[4];
  // 使用与输入相同的编解码器
  if (codec==0) {
```

```
    codec= getCodec(c);
  }

  // 打开输出视频
  return writer.open(outputFile, // 文件名
  codec,          // 编解码器
  framerate,      // 视频的帧速率
  getFrameSize(), // 帧的尺寸
  isColor);       // 彩色视频?
}
```

名为writeNextFrame的私有方法处理帧的写入过程（写入到视频文件或一系列图像）：

```
// 写输出的帧
// 可以是：视频文件或图像组
void writeNextFrame(cv::Mat& frame) {

  if (extension.length()) { // 写入到图像组

    std::stringstream ss;
    // 组合成输出文件名
    ss << outputFile << std::setfill('0') << std::setw(digits)
      << currentIndex++ << extension;
    cv::imwrite(ss.str(),frame);

  } else { // 写入到视频文件
    writer.write(frame);
  }
}
```

如果输出是独立的文件，就需要一个额外的获取方法：

```
// 设置输出为一系列图像文件
// 扩展名必须是.jpg、.bmp……
bool setOutput(const std::string &filename, // 前缀
  const std::string &ext, // 图像文件的扩展名
  int numberOfDigits=3,   // 数字的位数
  int startIndex=0) {     // 开始序号

  // 数字的位数必须是正数
  if (numberOfDigits<0)
    return false;

  // 文件名和常用的扩展名
  outputFile= filename;
  extension= ext;

  // 文件编号方案中数字的位数
  digits= numberOfDigits;
  // 从这个序号开始编号
  currentIndex= startIndex;

  return true;
}
```

最后在run方法的视频捕获循环中添加一个新的步骤：

```
while (!isStopped()) {

  // 读取下一帧（如果有）

  if (!readNextFrame(frame))
    break;

  // 显示输入帧
  if (windowNameInput.length()!=0)
    cv::imshow(windowNameInput,frame);

  // 调用处理函数或方法
  if (callIt) {

    // 处理帧
    if (process)
      process(frame, output);
    else if (frameProcessor)
      frameProcessor->process(frame,output);
    // 递增帧数
    fnumber++;

  } else {

    output= frame;
  }

  // ** 写入到输出的序列 **
  if (outputFile.length()!=0)
    writeNextFrame(output);

  // 显示输出的帧
  if (windowNameOutput.length()!=0)
    cv::imshow(windowNameOutput,output);

  // 产生延时
  if (delay>=0 && cv::waitKey(delay)>=0)
    stopIt();

  // 检查是否需要结束
  if (frameToStop>=0 && getFrameNumber()==frameToStop)
    stopIt();
 }
}
```

11.4.3 扩展阅读

在把视频写入文件时需要使用一个编解码器。编解码器是一个软件模块，用于编码和解码视频流。编解码器定义了文件格式和存储信息的压缩方案。很明显，用某种编解码器进行编码的视

频，必须用同一种编解码器才能解码。因此人们使用含有四个字符的代码来唯一地表示一种编解码器。这样，软件工具在写入视频文件之前，需要先读取这个四字符代码，以决定采用哪种编解码器。

编解码器的四字符代码

正如其名称所示，四字符代码是由4个ASCII字符组成的，拼在一起也可以转换成一个整数。用cv::VideoCapture打开视频文件，然后在get方法中使用CV_CAP_PROP_FOURCC标志，就能得到该视频文件的代码。我们可以在VideoProcessor类中定义一个方法，返回输入视频的四字符代码：

```
// 取得输入视频的编解码器
int getCodec(char codec[4]) {

  // 对于图像向量，本方法无意义
  if (images.size()!=0) return -1;

  union { // 表示四字符代码的数据结构
   int value;
   char code[4]; } returned;

  // 取得代码
  returned.value= static_cast<int>
    (capture.get(CV_CAP_PROP_FOURCC));

  // 取得4个字符
  codec[0]= returned.code[0];
  codec[1]= returned.code[1];
  codec[2]= returned.code[2];
  codec[3]= returned.code[3];

  // 返回代码的整数值
  return returned.value;
}
```

get方法总是返回一个double型数值，然后转换成整数。这个整数就是代码，可以用union数据结构从这个代码提取出4个字符。打开测试用的视频序列，然后使用以下代码：

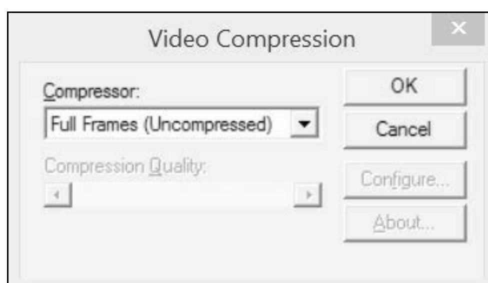
```
char codec[4];
processor.getCodec(codec);
std::cout << "Codec: " << codec[0] << codec[1] << codec[2] <<
  codec[3] << std::endl;
```

用上述语句可得到这个结果：

```
Codec : XVID
```

在写入视频文件时，必须用四字符代码指定编解码器。这就是cv::VideoWriter类open方法的第二个参数。可以使用与输入视频相同的代码（这是setOutput方法的默认选项）。也可以

传入值-1，该方法会弹出一个窗口，让用户选择可用的编解码器，如下图所示：



窗口中的列表所显示的就是该电脑中已经安装的编解码器。选中某个编解码器后，它的代码就会自动传给open方法。

11.4.4　参阅

- 网站https://www.xvid.com/提供了基于MPEG-4视频压缩标准的开源的视频编解码器程序库。还有一种Xvid的竞争者，名为DivX，它提供了专有但免费的编解码器和软件工具。

11.5　跟踪视频中的特征点

本章的内容包括对视频序列的读、写和处理。我们的目标是能够分析完整的视频序列。作为一个例子，本节我们来学习如何对视频序列做时序分析，以跟踪在帧之间移动的特征点。

11.5.1　如何实现

在启动跟踪过程时，首先要在最初的帧中检测特征点，然后在下一帧中跟踪这些特征点。因为我们处理的是一个视频序列，特征点所属的物体很可能会移动（这种移动也可能是由摄像机的运动引起的）。因此要找到特征点在下一帧的新位置，必须在它原来位置的周围进行搜索。这个功能由函数cv::calcOpticalFlowPyrLK实现。在函数中输入两个连续的帧和第一个图像中特征点的向量，返回新的特征点位置的向量。为了在整个视频序列中跟踪特征点，要一帧一帧地重复上述过程。因为在跟踪特征点时要穿越视频序列，不可避免地会丢失部分特征点，导致被跟踪的特征点数量逐渐减少。因此最好经常性地检测新特征点。

现在我们利用上节的框架来定义一个类，实现11.3节中介绍的FrameProcessor接口。这个类的数据属性包含检测和跟踪特征点所需的变量：

```
class FeatureTracker : public FrameProcessor {

  cv::Mat gray;          // 当前灰度图像
```

```
  cv::Mat gray_prev;     // 上一个灰度图像
  // 被跟踪的特征，从0到1
  std::vector<cv::Point2f> points[2];
  // 被跟踪特征点的初始位置
  std::vector<cv::Point2f> initial;
  std::vector<cv::Point2f> features; // 被检测的特征
  int max_count;     // 检测特征点的最大个数
  double qlevel;     // 检测特征点的质量等级
  double minDist;    // 两个特征点之间的最小差距
  std::vector<uchar> status; // 被跟踪特征的状态
  std::vector<float> err;    // 跟踪中出现的误差

  public:

  FeatureTracker() : max_count(500), qlevel(0.01), minDist(10.) {}
```

接下来定义process方法，它将在处理序列中的每个帧时被调用。一般来说，处理过程包含以下几个步骤。首先，如果需要就检测特征点。然后跟踪这些特征点，剔除无法跟踪或不需要跟踪的特征点，准备处理跟踪成功的特征点。最后，当前的帧和特征点作为下一个迭代项的上一帧和上一批特征点。下面是具体代码：

```
void process(cv:: Mat &frame, cv:: Mat &output) {

  // 转换成灰度图像
  cv::cvtColor(frame, gray, CV_BGR2GRAY);
  frame.copyTo(output);

  // 1. 如果必须添加新的特征点
  if(addNewPoints())
  {
    // 检测特征点
    detectFeaturePoints();
    // 在当前跟踪列表中添加检测到的特征点
    points[0].insert(points[0].end(),features.begin(),
      features.end());
    initial.insert(initial.end(),features.begin(),
      features.end());
  }

  // 对于序列中的第一个图像
  if(gray_prev.empty())
    gray.copyTo(gray_prev);

  // 2. 跟踪特征
  cv::calcOpticalFlowPyrLK(gray_prev, gray, // 2个连续图像
  points[0], // 输入第一个图像的特征点位置
  points[1], // 输出第二个图像的特征点位置
  status, // 跟踪成功
  err);   // 跟踪误差

  // 3. 循环检查被跟踪的特征点，剔除部分
  int k=0;
```

```
    for( int i= 0; i < points[1].size(); i++ ) {

      // 是否保留这个特征点?
      if (acceptTrackedPoint(i)) {

        // 在向量中保留这个特征点
        initial[k]= initial[i];
        points[1][k++] = points[1][i];
      }
    }

    // 剔除跟踪失败的特征点
    points[1].resize(k);
    initial.resize(k);

    // 4. 处理已经认可的被跟踪特征点
    handleTrackedPoints(frame, output);

    // 5. 当前特征点和图像变成前一个
    std::swap(points[1], points[0]);
    cv::swap(gray_prev, gray);
  }
```

这个方法利用了4个实用方法。如果要自定义跟踪功能，可以很方便地更换这些方法。第一个方法是检测特征点。我们在8.2节已经讨论过这个`cv::goodFeatureToTrack`方法：

```
  // 特征点检测方法
  void detectFeaturePoints() {

    // 检测特征点
    cv::goodFeaturesToTrack(gray, // 图像
      features,    // 输出检测到的特征点
      max_count,   // 特征点的最大数量
      qlevel,      // 质量等级
      minDist);    // 特征点之间的最小差距
  }
```

第二个方法判断是否需要检测新的特征点：

```
  // 判断是否需要添加新的特征点
  bool addNewPoints() {

    // 如果特征点数量太少
      return points[0].size()<=10;
  }
```

第三个方法根据程序定义的条件剔除部分被跟踪的特征点。这里我们剔除不移动的特征点（还有不能被`cv::calcOpticalFlowPyrLK`函数跟踪的特征点）：

```
  // 判断需要保留的特征点
  bool acceptTrackedPoint(int i) {

    return status[i] &&
```

```
    // 如果已经移动
    (abs(points[0][i].x-points[1][i].x)+(abs(points[0][i].y-
      points[1][i].y))>2);
  }
```

最后，第四个方法处理被跟踪的特征点，具体做法是在当前帧画直线，连接特征点和它们的初始位置（即第一次检测到的位置）：

```
  // 处理当前跟踪的特征点
  void handleTrackedPoints(cv:: Mat &frame, cv:: Mat &output) {

    // 遍历所有特征点
    for(int i= 0; i < points[1].size(); i++ ) {

      // 画线和圆
      cv::line(output,
        initial[i],  // 初始位置
        points[1][i],// 新位置
        cv::Scalar(255,255,255));
        cv::circle(output, points[1][i], 3, cv::Scalar
          (255,255,255),-1);
    }
  }
```

可以写一个简单的main函数，跟踪视频序列中的特征点：

```
  int main()
  {
    // 创建视频处理类的实例
    VideoProcessor processor;

    // 创建特征跟踪类的实例
    FeatureTracker tracker;

    // 打开视频文件
    processor.setInput("../bike.avi");

    // 设置帧处理类
    processor.setFrameProcessor(&tracker);

    // 声明显示视频的窗口
    processor.displayOutput("Tracked Features");

    // 以原始帧速率播放视频
    processor.etDelayetDelay(1000./processor.getFrameRate());

    // 开始处理
    processor.run();
  }
```

最终程序显示被跟踪的特征点随时间移动的过程。这里用两个不同瞬间的帧作为例子。这个视频中摄像机是固定不动，唯一的移动物体是年轻的骑车人。下面是处理完一些帧后得到的结果：



几秒钟后，得到下面的帧：



11.5.2　实现原理

要逐帧地跟踪特征点，必须在后续帧中定位特征点的新位置。假设每个帧中特征点的强度值是不变的，这个过程就是寻找如下的位移(u, v)：

$$\boldsymbol{I}_t(x,y)=\boldsymbol{I}_{t+1}(x+u,y+v)$$

其中I_t和I_{t+1}分别是当前帧和下一个瞬间的帧。强度值不变的假设普遍适用于相邻图像上的微小位移。我们可使用泰勒展开式得到近似方程式（包含图像导数）：

$$\boldsymbol{I}_{t+1}(x+u,y+v)\approx\boldsymbol{I}_t(x,y)+\frac{\partial\boldsymbol{I}}{\partial x}u+\frac{\partial\boldsymbol{I}}{\partial y}v+\frac{\partial\boldsymbol{I}}{\partial t}$$

可根据第二个方程式得到另一个方程式（根据强度值不变的假设，去掉了两个表示强度值

的项）：

$$\frac{\partial \boldsymbol{I}}{\partial x}u+\frac{\partial \boldsymbol{I}}{\partial y}v=-\frac{\partial \boldsymbol{I}}{\partial t}$$

这就是著名的光流约束方程。Lukas-Kanade特征跟踪算法使用了这个约束，同时又做了一个假设，即特征点邻域中所有点的位移量是相等的。因此我们可以将光流约束应用到所有这些位移量为(u, v)的点（u和v还是未知的）。这样我们就得到了更多的方程式，数量超过未知数的个数（2），因此可以在均方意义下解出这个方程组。在实际应用中，我们采用迭代的方法来求解。而且为了使搜索更高效且适应更大的位移量，OpenCV提供了在不同分辨率下进行计算的方法。默认的图像等级数量为`3`，窗口大小为`15`。当然这些参数是可以修改的。此外还可以设定一个终止条件，符合这个条件时就停止迭代搜索。`cv::calcOpticalFlowPyrLK`函数的第六个参数是剩余均方误差，用于评定跟踪的质量。第五个参数包含二值标志，表示跟踪对应的点是否成功。

上面描述了Lukas-Kanade跟踪算法的基本规则。具体实现时还做了优化和改进，使该算法在计算大量特征点的位移时更加高效。

11.5.3　参阅

- 第8章详细介绍了检测特征点的方法。
- B. Lucas和T. Kanade发表在*Int. Joint Conference in Artificial Intelligence*（1981，674-679页）的经典论文“An Iterative Image Registration Technique with an Application to Stereo Vision”描述了原始的特征点跟踪算法。
- J. Shi和C. Tomasi发表在*IEEE Conference on Computer Vision and Pattern Recognition*（1994，第593页至第600页）的“Good Features to Track”描述了原始特征点跟踪算法的改进版本。

11.6　提取视频中的前景物体

用固定位置的摄像机拍摄时，背景部分基本上是保持不变的。这种情况下，我们关注的是场景中移动的物体。为了提取这些前景物体，我们需要构建一个背景模型，然后将模型与当前帧做比较，检测出所有的前景物体。这正是本节要实现的内容。前景提取是智能监控程序的基本步骤。

如果有该场景的背景图像（即没有前景物体的帧）供我们使用，那么提取当前帧的前景物体就会非常容易，只需要比较两个图像：

```
// 计算当前图像与背景图像之间的差异
cv::absdiff(backgroundImage,currentImage,foreground);
```

每个差异足够大的像素都可作为前景像素。但是在大多数情况下背景图像是很难获得的。实

际上很难保证一个图像中没有任何前景物体，并且在繁忙的场景中，这种情况是极少出现的。并且由于光照条件变化（如从日出到日落）、背景中物体的增加或减少等原因，背景也会随着时间变化。

因此有必要动态地构建背景模型。实现方法是观察该场景并持续一段时间。如果我们做一个假设：在每个像素位置，背景在绝大部分时间都是可见的，那么建立背景模型的方法就很简单，只需计算所有观察结果的平均值。但这种做法其实并不可行，原因有多个。首先，在计算背景之前需要存储大量的图像。第二，在为计算平均值而累计图像的时候，是无法提取前景的。这种解决方案还需要考虑几个问题，即为了计算可靠的背景模型，需要累计何时的、多少数量的图像。另外，如果有些图像中的某个像素正在监视一个前景物体，那么它们就会对计算平均背景产生很大的影响。

更好的策略是用定时更新的方式，动态地构建背景模型。实现方法是通过计算滑动平均值（又叫移动平均值）。这是一种计算时间信号平均值的方法，并且该方法考虑了最新收到的数值。假设p_t是时间t的像素值，μ_{t-1}是当前的平均值，那么要用下面的公式来更新平均值：

$$\mu_t = (1-\alpha)\mu_{t-1} + \alpha p_t$$

其中参数α称为学习速率，它决定了当前值对计算平均值的影响程度。这个值越大，滑动平均值对当前值变化的响应速度就越快。为了构建背景模型，必须在新的帧到达时对每个像素计算滑动平均值。然后就可以根据当前图像与背景模型之间的差异，判断一个像素是否为前景像素。

11.6.1 如何实现

我们创建一个用滑动平均值动态构造背景模型的类，并通过减法运算提取前景物体。这个类需要有以下属性：

```
class BGFGSegmentor : public FrameProcessor {

  cv::Mat gray;         // 当前灰度图像
  cv::Mat background;   // 累积的背景
  cv::Mat backImage;    // 当前背景图像
  cv::Mat foreground;   // 前景图像
  // 累计背景时使用的学习速率
  double learningRate;
  int threshold;        // 提取前景的阈值
```

主要处理过程包括将当前帧与背景模型作比较，然后更新该模型：

```
// 处理方法
void process(cv:: Mat &frame, cv:: Mat &output) {

  // 转换成灰度图像
```

```
  cv::cvtColor(frame, gray, CV_BGR2GRAY);

  // 采用第一帧初始化背景
  if (background.empty())
    gray.convertTo(background, CV_32F);

  // 背景转换成8U类型
  background.convertTo(backImage,CV_8U);

  // 计算图像与背景之间的差异
  cv::absdiff(backImage,gray,foreground);

  // 在前景图像上应用阈值
  cv::threshold(foreground,output,threshold,255,cv::
    THRESH_BINARY_INV);

  // 累积背景
  cv::accumulateWeighted(gray, background,
    // alpha*gray + (1-alpha)*background
    learningRate,  // 学习速率
    output);       // 掩码

}
```

使用自定义的视频处理框架，可以这样构建前景提取程序：

```
int main()
{
  // 创建视频处理类的实例
  VideoProcessor processor;

  // 创建背景/前景的分割器
  BGFGSegmentor segmentor;
  segmentor.setThreshold(25);

   // 打开视频文件
   processor.setInput("bike.avi");

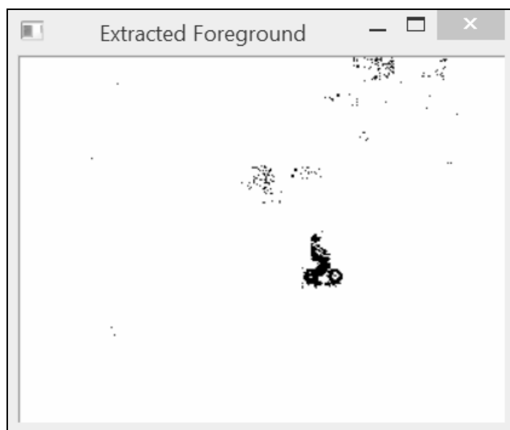
  // 设置帧处理对象
  processor.setFrameProcessor(&segmentor);

  // 声明显示视频的窗口
  processor.displayOutput("Extracted Foreground");

  // 用原始帧速率播放视频
  processor.setDelay(1000./processor.getFrameRate());

  // 开始处理
  processor.run();
}
```

最后得到一些二值前景图像，其中一个如下图所示：



11.6.2 实现原理

用`cv::accumulateWeighted`函数计算图像的滑动平均值非常方便，它在图像的每个像素上应用滑动平均值计算公式。注意，作为结果的图像必须是浮点数类型的。正因为如此，在我们将背景模型与当前帧做比较之前，必须先把它转换成背景图像。对差异绝对值（先用`cv::absdiff`计算，然后用`cv::threshold`）进行阈值化，以提取前景图像。然后把这个前景图像作为`cv::accumulateWeighted`函数的掩码，以避免修改已被认定为前景的像素。之所以要这么做，是因为在前景图像中，已被认定为前景的像素值为`false`，即`0`（这也是结果图像的前景物体显示成黑色的原因）。

最后需要注意，为了简化，我们在构建背景模型时采用了被提取帧的灰度图像。如果构建彩色背景，就需要在多个颜色空间下计算滑动平均值。不过在刚才介绍的方法中，主要的难点在于，针对特定的视频，如何选择合适的阈值以得到满意的结果。

11.6.3 扩展阅读

上述提取前景物体的方法比较简单，它适用于背景相对固定的简易场景。但是在很多情况下，背景的某些部位会在不同的值之间波动，导致背景检测结果频繁出错。产生这种现象的原因，有移动的背景物体（如树叶）、刺眼的物体（如水面），等等。物体的阴影也会产生问题，因为阴影是会移动的。为解决这些问题，我们引入了更复杂的背景模型。

混合高斯模型

高斯混合方法是这些改进型算法的一种。它的处理方式与前面介绍的基本一致，但做了几项改进。

首先，该方法适用于每个像素有一个以上的模型（即一个以上的滑动平均值）的情况。这

样，如果一个背景像素在两个值之间波动，那么就会存储两个滑动平均值。只有新的像素值不属于任何一个频繁出现的模型，这个像素才会被认定为前景。模型的数量可以在参数中设置，通常为5个。

第二，每个模型不仅保存了滑动平均值，还保存了滑动方差。它的计算方法如下：

$$\sigma_t^2 = (1-\alpha)\sigma_{t-1}^2 + \alpha(p_t - \mu_t)^2$$

计算得到平均值和方差后用于构建高斯模型，根据高斯模型可计算某个像素值属于背景的概率。改用概率而不是绝对差值来表示后，阈值的选择就会更加容易。采用这个模型后，如果某个区域的背景波动较大，就需要有更大的差值才能被认定为前景物体。

最后，如果某个高斯模型满足条件的概率不够高，它就会被排除在背景模型之外。反之，如果发现一个像素值在当前背景模型之外（即为一个前景像素），那么就会创建一个新的高斯模型。如果随后这个新建的模型满足条件了，就把它作为正确的背景模型。

比起前面的前景/背景分割法，这个算法更加复杂，实现起来更加困难。幸好OpenCV已经有了现成的类，名为`cv::BackgroundSubtractorMOG`，它是`cv::BackgroundSubtractor`的子类，后者的通用性更强。如果采用默认参数，使用这个类就变得非常简单：

```
int main()
{
  // 打开视频文件
  cv::VideoCapture capture("bike.avi");
  // 检查打开视频是否成功
  if (!capture.isOpened())
    return 0;
  // 当前视频帧
  cv::Mat frame;
  // 前景的二值图像
  cv::Mat foreground;
  cv::namedWindow("Extracted Foreground");
  // 混合高斯模型类的对象，全部采用默认参数
  cv::BackgroundSubtractorMOG mog;
  bool stop(false);
  // 遍历视频中的所有帧
  while (!stop) {
    // 读取下一帧（如有）
    if (!capture.read(frame))
      break;
    // 更新背景并返回前景
    mog(frame,foreground,0.01)
    // 学习速率
    // 改进图像效果
    cv::threshold(foreground,foreground,
      128,255,cv::THRESH_BINARY_INV);
    // 显示前景
```

```
        cv::imshow("Extracted Foreground",foreground);

        // 产生延时，或者按键结束
        if (cv::waitKey(10)>=0)
          stop= true;
      }
    }
```

在代码中，只需创建这个类的实例并调用它的一个方法，这个方法更新背景并返回前景图像（额外的参数是学习速率）。另外，这里计算的背景模型是彩色的。OpenCV实现的方法还包含了排除阴影的机制，其原理是检查亮度的局部变化是否为像素值变化的唯一原因，或者是否包含了色度的变化。

此外还有该模型的第二种实现方法，称为`cv::BackgroundSubtractorMOG2`。它所做的一个改进是，动态地确定每个像素上高斯模型的数量。上述例子中可用这个类替代原来使用的类。可以针对一些视频使用这两种不同的方法，观察它们各自的性能。一般来说，使用`cv::BackgroundSubtractorMOG2`会快得多。

11.6.4 参阅

- C. Stauffer和W.E.L. Grimson发表在*Conf. on Computer Vision and Pattern Recognition*（1999年）的论文“Adaptive Background Mixture Models for Real-Time Tracking”更完整地描述了混合高斯算法。

版权声明

计算机视觉是机器准确识别、理解和表示信息，从而感知并与世界交互的媒介，在人脸识别、智能驾驶、手势游戏、图像搜索、自动定位等各领域都发挥着极为重要的作用。OpenCV作为开源程序库，提供了500多个用于图像和视频分析的优化算法，是相关领域研发人员的主要开发工具。

本书不仅基于OpenCV库介绍了计算机视觉编程，让零基础用户能全面掌握各种开发方法，更可贵的是注重实用性。作者不单纯罗列各种功能，而是逐一讲解了如何使用各种功能完成特定任务。你将在学习概念、原理、示例的过程中了解50多个实例的具体应用，学会快速构建真正满足需要的视觉应用。作为升级版，本书全面修订、升级了所有实例，也增加了对OpenCV功能的介绍，覆盖的知识点更加全面。

本书不仅适合有C++背景的计算机视觉新手学习，也是专业开发人员必备的参考，还可作为相关专业学生的辅助教材。

本书内容：

- OpenCV库安装和程序开发；
- 通过操作像素来处理图像；
- 用直方图分析图像；
- 分割图像为同质区域，并提取有意义的目标；
- 用滤波器增强图像内容；
- 利用图像几何关联同一场景不同视角的图像；
- 利用不同角度的图像校准相机。



[PACKT] PUBLISHING

图灵社区：iTuring.cn

热线：(010)51095186转600

分类建议 计算机/程序设计/OpenCV

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-39850-5

定价：59.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们:

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks